S.R.43

THE ENGLISH ELECTRIC CO., LTD.

NELSON RESEARCH LABORATORIES
STAFFORD
MATHEMATICS DEPARTMENT.

Telephone:—Stafford 700.

Report No. NS y 80
Date 25.11.57.
Reference
Order No.

Front Sheet.
Data Sheets 1-35.

PREPARING AND TESTING DEUCE PROGRAMMES.

Report by
P. J. Landin

SUMMARY:

This report describes the various stages of writing and testing a DEUCE program with special emphasis on procedures that make a program easy to check, use and modify. The report has been specially prepared for distribution to members of the December, 1957, DEUCE Programmers Course.

pp P. J. Landin.
MATHEMATICS DEPARTMENT.

[illegible]

RF

CONTENTS

1. INTRODUCTION

Much of the programmers' and computers' time spent in testing and altering programs can be saved if more attention is paid to their detailed design before writing them and to the detailed test requirements before checking them on the computer.

This report records information that is at present mainly learned by bitter experience, often several repetitions of the same bitter experience, and finally accepted as so obvious as to be not worth telling newcomers to the craft.

It displays the available alternatives in many of the decisions facing the programmer with special emphasis on producing programs that are easy to check, use and modify.

2. WRITING A PROGRAM:

This chapter describes some of the decisions a programmer will (despite himself) have made by the time his program is written.

The best time to make them is sometime after writing the first draft of the logical flow diagram and before writing the computer flow diagram.

Most of the procedures suggested here are aimed at making a program easy to diagnose faults in, easy to modify and easy to use. These aims rarely conflict with one another or with rapid preparation of a program since the extra time spent in designing a program will be more than repaid by avoiding the lengthy job of salvaging a badly designed or undesigned program.

There are some applications of DEUCE to which these recommendations may not all apply in detail, but it is intended to lay out an approach to programming that will have universal application, and illustrate it with references to DEUCE programs.

2.1. Inputting and Obeying the Instructions.

2.11. General Procedure.

2.11.1. For a self-contained program pack.

Each computer operation starts by establishing a standard computer state. This is partly achieved by the operator (see "Standard Operating Instructions" by A. Birchmore) and partly by instruction cards, in the following way. A program that does not use any computer-stored data starts with the three cards:-

Type I Initial Card (to establish correct mc number).

Clear Drum ZP13/1 (to write zeros in all drum addresses and leave both sets of heads at position zero).

Set Clock Track ZP 36 (to write on 15/15 a clock track with which Sync Clock Track ZP37 and Clock Track Set or Sync ZP34 and Post Mortem ZP29 can all synchronise).

These are followed by cards containing instructions to be stored in the computer (together with instructions that will store them there). The cards to do this are any or all of

(1) Read to Drum ZP14.
Triads of information destined for tracks of the drum except track 15/15.

(2) Fill Short Tanks etc. ZP35.
Triads of information destined for short tanks, OPS, triggers and head positions.

(3) Triads for DL's 12 to 1.

The second of these items is too complicated to use, except in an emergency to reconstruct a particular computer state. The third of them can often be advantageously dispensed with if one of the schemes described in 2.2. to 2.5. is adopted.

2.11.2 For a Program pack that uses computer-stored data.

A program pack that uses computer-stored data will generally be read by instructions initiated by a program already in the computer, in which case the first three cards listed in 2.11.1 should be omitted.

In an emergency it may be necessary to break in on the current computer operation and the pack should then be preceded by Clock Track Set or Sync ZP34 instead of the three cards listed in 2.11.1. (If mercury-stored information is required then the operator must clear TS COUNT manually and use the run in key instead of the initial input key.)

2.12. Dividing a Program into Sections.

There are 402 mercury addresses in DEUCE of which 256 can be used as next instruction addresses, and many programs require a lot more than 400 DEUCE instructions. It is therefore often necessary to overwrite instructions by other ones during the course of a program. The overwriting program can be stored on cards or on the drum, and the instructions to transfer it to the mercury must be already in the mercury.

Transferring 32 instructions to a delay line takes nearly a second from cards and up to 50 ms from the drum, and 32 instructions can take as little as 2 ms to obey, so it can save a lot of time if the program is partitioned in such a way that instructions are obeyed as often as possible without being overwritten; i.e. it is more efficient to overwrite programs in between loops than in the middle of them. It also follows from these figures that if there is room on the drum to store all the program there is little to lose by initially transferring the entire program from cards to drum (via mercury) and then transferring from drum to mercury as required.

The gains are:-

(a) even if some instructions do have to be transferred to the mercury on several different occasions, they only have to be read from cards once and this is by far the longer transfer;

(b) a card pack can be made containing all the instructions with only one copy of each part of the program and without the necessity to interleave data with instructions each time the program is used. This simplifies the operating instructions and reduces the chance of operating errors;

(c) the whole program or any part of it can be made re-entrant or can easily be adapted as part of an even larger program if later required.

Efficient partitioning usually means that each item of the overall logical flow diagram is obeyed without interruption from program transfers and so the program consists of several sections each with a specific function, each working on numbers read from cards, and/or left in the computer by previous sections, and each leaving results in the computer for succeeding sections and/or on cards.

If furthermore each section is as self-contained as possible and a precise specification of what it does is given then

(d) programming time can often be saved by using existing programs, e.g. for input and output of matrices, in computations to which they are not obviously relevant;

(e) it is easier to localise mistakes and to make alterations that are only local in effect.

2.13 Standard Program Sections.

In 2.14 a standard form for a program section is described that has the following additional advantages:-

(a) All the transfer of program from cards to drum and from drum to mercury can be done with control programs that are already written.

(b) These control programs have built-in program testing facilities, e.g. for tracing the progress of a large program, restoring control if the program runs amok, re-starting at any required section, and re-writing a single track during testing.

(c) The sections can be written so that they may be obeyed indifferently from the drum or the reader. So if for a particular use of the program there is not room on the drum for data and instructions, a small re-arrangement will often permit the instructions to be obeyed directly from the reader without requiring any drum storage.

2.14 The Rules for a Standard Program Section.

The following conditions do not greatly restrict the programmer and allow him to use any of the control programs described below:-

(a) Each section of program is transferred to consecutive DL's including DL1 before it is obeyed. If it is stored on the drum it is stored on consecutive tracks (lowest DL number in lowest track number). It may have up to 10 DL's(although instructions in 9 and 10 will have to be transferred again before they can be obeyed.) The cards are punched with normal initial instructions except for DL1 which has

```
          blank
1, 0-1   (1) 26, 25 X
1, 0-1       30, 31 X
1, 9-24       0, 29 X
```

The instruction 9-24 x punched on the 4th row of the DL1 triad is obeyed in mc 31 if the section is obeyed from the reader. The last row of the triad is punched with the section number of the program at P17. This is only transferred from the card to the computer when the section is not obeyed directly from the reader. It serves during testing to identify easily the section currently being obeyed from the drum.

(b) Each section starts

(1_{31} 9-24 X) (only obeyed if the section is obeyed directly
1_{30} from the reader.)

and finds in the QS's (and in certain other short tanks depending on the control program used) and DL's, except DL11 and the ones it occupies, whatever was left there by the last section.

(c) Each section can use any of the stores except 12_{29-31}, track 15/15, and the tracks containing program sections or the control program. So the copy of the program on the drum is left unchanged (unless replenished from the reader) and each transfer of a particular program section to the mercury places an identical set of instructions there.

(d) Each section ends by placing a parameter that specifies what happens next and leads out with

$$\begin{array}{l} \quad 12\text{-}1 \quad (32) \\ 1_{30} \end{array}$$

and leaves the contents of the QS's (and of certain other short tanks depending on the control program used) and DL's except DL11 to be found by the next section (which will overwrite at least DL1.)

The form of the parameter, and which of the short tanks are preserved, depends on which control program is being used.

2.15 Fixing the Order in which Sections are Obeyed.

The way in which the order of obeying the sections is specified depends on the context and on the control program used:

(a) The simplest is that each section ends by placing a parameter saying which section comes next. It might have a choice of exit points specifying different successors depending on discriminations made while it is obeyed. (ZC01/3 ZC11 and ZC13 all allow this.)

(b) One of the sections may use another section as a subroutine, specifying not only which section is to be used next, but also the re-entry point to itself afterwards (ZC11 allows this.)

(c) There may be a master routine which uses all the other sections as subroutines and does nothing itself except this (ZC01/3 and ZC11 allow this. For ZC01/3 this master routine is simply a list of the section numbers in the order of execution with special facilities for loops.)

If all the basic program sections you require are already written and all you want to do is to string them together then you should use the master routine method.

If you are writing nearly all the program yourself and the logical arrangement of sections is fairly simple, method (a) is the best with perhaps slight modification of the ends of existing sections to make them lead correctly to their successors.

If the logic is more complicated, with sections using other sections as subroutines and re-entering themselves, ZC11 must be used.

It is possible approximately to rank the three control programs mentioned in various ways:

By power and flexibility	ZC01/3 (but rather inconvenient for using sections as subroutines) ZC11 ZC13
By simplicity in use	ZC01/3 ZC13 ZC11
By efficiency in use of computer time and storage once the program is prepared and tested.	ZC13 ZC11/ ZC01/3 (threshold time of approximately 1 sec per section).

For further information on ZC013 see DEUCE News 10
on ZC11 see R.A.E. Technical Note MS 31 "The Assembly of Large Programs for the Automatic Computer DEUCE".
on ZC13 see Library Program Report.

2.2. Input of Data

2.21. Punched Data.

Some programs require that each time they are used certain parameters are specified. Even if only one use is to be made of it a program can often be tested more effectively by varying these. No program should require parameters to be punched among its instructions each time it is used since this complicates pack assembly and provides opportunities for mistakes during both testing and use.

Nor should they be put into the computer via the I.D. Lamps or any other manual input since this is slow and provides opportunities for unrecorded operating mistakes. Nor should they be read from cards punched with "initial instructions" used for reading instructions into DL's, since this complicates data preparation and compels the use of binary. Instead the extra instructions necessary to read them from cards (preferably decimally punched) should be written and these cards prepared with other punched data each time the program is used.

A programmer must program not only the computer, but also the people who will use his work (including himself) and his job includes writing the instructions for preparing data and assembling the card pack (see 3.1). This should be done before finalising the form of the program. It is frequently worth while to simplify data preparation at the expense of program complication. Decimal is generally better than binary and the logically simple arrangement of numbers with no unnecessary information except checked redundancy checks (see 2.42) may be better than the arrangement that is easiest to program (as a trivial example, do not require data to be punched with P 54's).

The only exception to these rules is that sometimes the operator is required to give the computer information resulting from decisions made during the running of the program as a result of the computer's behaviour. The use of manual input and the circumstances that may make it unavoidable are considered in 2.22.

2.22 Manual Input

A program should be so written that the operating instructions are as simple as possible with minimal opportunity for mistakes, especially irrevocable mistakes. Complicated instructions result in computer time being wasted while the operator unravels and obeys them or, worse, in delays in getting jobs done because they are disobeyed. In 3.5 the even more important matter of making them precise will be discussed.

The sorts of action required of an operator are

(a) pressing keys and turning knobs

(b) inserting at specified places in the pack of cards to be read, cards that have been produced by the computer

(c) punching cards and inserting them in specified places in the pack of cards to be read

(d) recording on paper information obtained by observation during the running of the program.

These may be unconditional instructions or conditional on certain behaviour of the computer. They will now be discussed one sort at a time to see if they can be eliminated.

(a) Obviously not all Key work (e.g. pressing the initial input key) can be avoided. Complicated instructions such as setting up a large binary number on the I.D. keys can be replaced by punching it on a card, which is preferable since it is recorded. A particular sort of knob-turning that could almost always be eliminated, but often with great programming inconvenience is the use of the punch numbering switches (4 counters, 8 fixed switches and associated keys). It can happen that the rules for operating them involve periods of idle computer time in excess of actual computing time. This occurs especially when a program originally intended for large values of some of its parameters is used with small values of them.

(b) These instructions can only be reduced to the extent that internal storage is available.

(c) If precise operating instructions are possible then this sort can be replaced by computer instructions and thus punching is eliminated. If not, then not.

(d) Again if precise operating instructions are possible this sort could be replaced by punched output but possibly at the expense of complicating the subsequent card processing to an unreasonable extent (e.g. if special cards have to be removed from the output pack before printing.)

Any unpunched information indicated by the operator to the computer or given by the computer to the operator, should be coded as simply as possible. For example, if a choice from four alternatives is to be indicated they should be coded 0,1,2,3 or 1,2,3,4 rather than 5,17,18,31. Furthermore, in the case of signals from the operator to the computer it should be made physically impossible to give an irrevocably inappropriate signal. Particular cases of this are that no program should rely on the I.D. lamps being the same on two consecutive glances at them or be affected by the I.D. lamps or (during calculation) the T.I.L. key except in ways that are known and declared in the specification.

The operating described above is the operating that is planned as part of the program's actual use, not the operating that the programmer does during testing. For example, a program that does not use manual input (other than essentials such as initial input key) or visual output is easier to operate in use and consequently easier to test, but this does not preclude the use of these devices during testing (see 5 and 6.)

2.3. <u>Intermediate Results</u>.

2.31 <u>Computer-Stored Intermediate Results</u>

Each section of program is given numbers (including parameters, triggers, etc.) either by its predecessors or by the operator, and produces results either for its successors or for the operator. All these numbers except what are originally given to the computer on cards and what are finally taken away for subsequent processing are intermediate results of the current computer operation.

The time taken to obey a section is made up of the time to transfer the instructions to the mercury, the time to transfer its data and results to and from the mercury if this is necessary, the access time to instructions and number stores in the mercury and the productive computing time.

So other things being equal it saves computer time to reduce the number of program transfers by operating on as many numbers as possible with one section while it is in the mercury. Also other things being equal it saves computer time to reduce the amount of number transfers by operating with as many sections as possible on the same numbers keeping intermediate results in the mercury. Since these usually conflict, other things are rarely equal and the decision is sometimes difficult. Three cases are worth special attention;

(a) If the sizes of the batches of intermediate results are parameters of the calculation (as in matrix operations or evaluating formulae for several cases) then using mercury storage limits these parameters much more severely than using drum storage, especially if the drum addresses are programmed parametrically and can be easily fixed at a later stage or even calculated by the program. This is useful if the most efficient use of the store depends on more than one independent parameter.

(b) If the access to intermediate storage is mainly consecutive then much less time is lost on drum transfers than if it is widely scattered. There are in the library subroutines for referring to one or more consecutive strings of drum addresses via buffer DL's.

(c) If the computer flow diagram is logically simple with only very small and very large loops (if any) then by using very small sections most of the mercury store can be released for intermediate results. In this case access to instructions on the drum can be consecutive and done directly without reference to a drum-stored control program. This case usually arises when a great deal of rather patternless computing is being done with a moderate amount of unhomogeneous data (never exceeding say a few hundred numbers.)

2.32 Auxiliary External Storage for Intermediate Results.

Intermediate results may be stored on punched cards. This can arise in three ways:-

(a) There is not room in the computer to store all the information it must remember at a certain stage of the calculation. Alternatively, there would not be in the largest cases the program can deal with, and so it uses card storage indiscriminately in all cases.

(b) The computer is required for quite different work between producing and requiring the intermediate results. The time lag might be imposed by outside circumstances or might be to allow other (final) results to be examined.

(c) The program uses previously written programs whose data or final results are punched but are merely intermediate data in the current operation (e.g. card to card binary-decimal or decimal-binary conversion.)

Causes (b) and (c) may force the use of "operator storage" for intermediate results by outputting visually an item of information that must be input manually at a later time. (e.g. how far a job that is being temporarily broken off has got.) This should be avoided wherever possible.

2.4. Programmed Checks.

2.41. Programmed Checks on the Appropriateness of Data.

Most programs are only intended to work on certain special cases of the data that one might conceivably give them. For example a program may read and store decimally-punched data in batches of a size parametrically specified and then perform an iterative calculation on them. If the cards are nonsensically punched, if a batch is too large to fit on the drum, if some parameters that should be equal are not, if the operator lights the P.32 lamp when the program required P1, P2 or P3 to specify a three-way choice, if the numbers are such that an intermediate result exceeds capacity or the calculation involves dividing by zero, or if they fall outside the range of convergence of the iterative procedure then something will go wrong. Either the program will interpret inappropriate data as best it can and unapologetically produce meaningless answers, or it will misbehave (by going into a loop or stopping) in a rather random way, or it will give some result that can be interpreted by reference to its specification as meaning "nondecimal punching", "incompatible dimensions", etc.

This last alternative is much the most desirable of the three since it helps the operator or user to trace the (possibly off-computer) error that caused it. So a program specification should say what its outcome is for any data whatsoever, however inappropriate.

Just as a program may test the appropriateness of its data, a part of a program or a single section or subroutine may do the same. In this case data may consist not only of cards and manual input but also of intermediate computer-stored numbers. The above remarks apply to these checks as well except that the fault about which they can supply diagnostic evidence may be in a previous part of the current computer run, rather than in a precomputer operation like copying or hand-punching.

The outcome of a program should be specifiable not merely for the case of one data error but forcombinations of them. All these can be deduced from the overall and section logical flow diagrams which should be so designed that the most useful thing happens in each case. For example a store check on an operand should if possible be tested before testing the arithmetic of the operation in question, since inappropriate data might cause an arithmetic ~~failure but not vice~~ versa. check to fail, but wrong arithmetic is less likely to cause a data check to fail.

2.42. Programming checks on the off-computer and computer operators.

The checks just described are only possible where there is redundant information in the data. The preventitive and diagnostic use of checks is so great that redundancy is sometimes introduced into data just so that it can be checked. Wherever possible a calculation should be checked by an independent calculation and storage checked by a sum check. Linear operations provide many opportunities for distributive checks that take a negligible amount of extra time, but with some calculations it is difficult to do more than "see if the answers look sensible".

A check sum is usually a nonsense sum of words (or smaller groups of digits) added together without regard for meaningfulness modulo the range of the group of digits. It can be computed each time a batch of numbers is produced and then checked against a freshly computed sum whenever the batch is used. (The particular case in which the group of digits is one bit is calleda parity check.)

A check sum prepared with the earliest written copy of a set of numbers and carried through and checked at each stage of copying, punching and transferring to the computer store can be used to detect errors both off and on the computer.

While it is true that every redundancy check on appropriateness of data could also be failed owing to a computer fault, there are some redundancy checks that <u>can only</u> be failed because of a computer fault. But this is only true if previous checks have prevented anything but appropriate data getting to this point. For example, a sum check may be failed because part of the batch of numbers summed has since been overwritten owing to the inappropriate length of another batch. But if that and all things like it were previously guarded against the sum check would be entirely a check on the operation of the computer.

2.43. <u>Failure Action.</u>

The best thing for the computer to do after failing a check is automatically repeat the faulty operation, but this is not possible if the necessary intermediate results have been overwritten. At the worst the computer can loop back to the branch instruction that caused the failure. This has several advantages,

(a) it is quite determinate

(b) the operator can at least discover what is the <u>direct</u> cause of failure (e.g. 14 and 15 being unequal)

(c) during testing it can be passed by manual intervention.

In any case some indication of what has happened should be made and if this is visual the operating instructions should say what the operator is to do about it. The standard visual failure indications are described in 2.52.

2.44. <u>The use of Programmed checks during testing.</u>

Check failures will be much more noticeable to the programmer as clues to programming errors than in their ultimate purpose. This alone justifies some care in making the checks comprehensive and mutually independent.

2.5. <u>Output</u>

2.51. <u>Punched output.</u>

Just as a programmer must consider how the data is to be prepared so must he consider what subsequent processing is necessary. Again more complicated programming may be justified by less complicated off-computer operations. In particular all the information to be produced by the computer when the program is used should be punched on cards in a printable form.

2.52. Visual (and Aural) Output.

The only exception to this is information which the operator must act on while the program is still running. This information must be output distinctively and simply via the IS and OPS lamps, the buzzer and sometimes the sequence of reading and punching cards.

The most frequent use is for failure indications. Despite the difficulty mentioned in 2.42. of classifying checks according to their function certain conventions exist.

Visual failure indications use the IS lamps. The computer reaches some characteristic stopped instruction and after a one-shot does the most useful thing possible in the circumstances, e.g. repeating the incorrect operation or at worst returning to the branch instruction that did the check. Approximately, the buzzer is sounded by computer failures (i.e. computer sum checks, arithmetic checks and other checks on redundancy artificially introduced in the computer) and not for data failures (i.e. on sum checks and other checks on redundancy artificially introduced outside the computer and also on natural redundancy arising from the possibility of providing inappropriate data).

The stopped instructions in common use are:-

29-31 X	for a computer sum check failure
24-31 X	for an arithmetic failure
n-29 X	for a punched sum check failure (discrepancy in n)
31-29 X	for incompatible parameters.

3. DOCUMENTING A PROGRAM.

There is more to producing a computer program than writing and coding a flow diagram of instructions that works. It must be recorded in a way that makes it easy for anyone (including the programmer in six months time) to use or modify. The information needed for this is listed below. There are three reasons why it should be written down before rather than after starting to test the program.

(a) The act of documenting the program frequently draws attention to unsatisfactory or incorrect features.

(b) It gives the programmer a better overall picture of his own work.

(c) The resulting documents are extremely useful in checking the program and localising and successfully correcting mistakes detected during testing.

3.1. Documents relating to the Program as a whole.

3.11. Specification.

Given a particular program together with its operating instructions (and provided that they are both unambiguous in outcome, which they may not be) and a stopped computer, the following things may affect the outcome when it is obeyed.

(a) The initial state of the computer; that is the contents of its 8595 stores (including TS COUNT) the ID, OPS, triggers head positions and states of the reader and punch.

(b) The punching (in sensed columns) on any cards (other than the program cards) that are previously prepared and assembled ready for putting in the reader hopper.

(c) The expressed intention of the user, if this is invoked by the operating instructions (e.g. the operator may be required to put on the P1 ID lamp if intermediate results are required.)

Usually quite a lot of (a) (and maybe of (b) and (c)) is irrelevant to the outcome.

The outcome consists of:-

(d) The final state of the computer (including TS COUNT which may be important since it determines, with other things, for example, whether the program is re-entrant.)

(e) The punching on output cards (including that produced by the punch switches.)

(f) Any information (such as the appearance of lights on the control panel or the rate of punching cards) that may be collected by the operator during the running of the program as a result of following the operating instructions.

Usually quite a lot of (d) and (f) (and maybe of (e)) is not of interest since it does not affect the course of subsequent events inside or outside the computer.

The specification of a program defines (the interesting parts of) the outcome for certain particular forms of initial state and/or input and/or users intentions. For example a program for solving linear equations by successive condensation about the largest possible pivot might produce the solution, or a measure of bad behaviour of the given matrix, or diagnostic indications of faultily punched data, according to the data that it is given.

So the specification has two parts. First the sort of data for which it will produce satisfactory results and specifiable unsatisfactory results (these should together be as exhaustive as possible.) Second the outcome for any such data.

3.12. The overall logical flow diagram.

The overall logical flow diagram is a flow diagram in which each item is either one section of the program together with the questions by which it chooses its successor or else a question asked with the same purpose in the master routine if there is one (see 2.14).

It ties together the other information relating to the program as a whole with that relating to each section (see 3.2.) For example any two sections that are consecutive here must have precisely matching final and initial conditions as stated in their individual specifications.

3.13. Program Pack List.

The program pack list is a complete list of the cards to be put into the computer that are not specially prepared for each use. It identifies in order all the cards by referring to card numbers and routine numbers of which complete details are given in the coding or in the description of previously written routines that are being used. It specifies any cards (including initial cards and program parameter cards). not described in the coding sheets of the various sections.

It might simply be card 0, Type I, Initial Card

cards 1-15 (see coding)

but could be longer if many library routines, some of them slightly modified, were used.

3.14. Pack Assembly Instructions.

The cards put into the reader hopper are of the following kinds:-

(a) program pack other than cards requiring special punching for each use (a requirement to be sternly avoided.)

(b) data cards and all other cards in which punching is specially done preparatory to each use of the program.

(c) cards produced by the computer during the current operation.

(d) cards punched by the operator as a result of decisions based on the behaviour of the computer during the current operation.

The pack of cards to be passed through the reader is assembled in trays so that they can be loaded into the hopper and removed from the stacker precisely in the assembled order without regard to where the pack is split. The position where cards of sort (c) and (d) might be inserted during the operation are marked with identification of these cards (by referring to the operating instructions, which include some description of what the computer does; see 3.15.)

3.15. Operating Instructions.

The instructions to the operator are about

(a) when to press and turn which keys and knobs which way.

(b) which output cards if any to insert into the reader pack at marked places.

(c) what to punch on what cards during the running of the program (to be inserted into the reader pack at marked places)

(d) what information about the running of the program to record.

It sometimes happens that a program is intended to do one of several things at the discretion of the user and indicated by manual control. Such a device should if possible be replaced by a punched indication that can be previously prepared. If it is not then some operating instructions will be conditional on the user's previously expressed intention e.g. "If intermediate results are required put P1 on the ID after the 1st set of cards have been punched." They may also be conditional on the outcome of the program, e.g. on failure indications being given.

Just as it is possible to write a program that has no determinate outcome (e.g. by disobeying the reader timing rules) so it is possible to write imprecise operating instructions that have no determinate outcome. Both these should be avoided. Operating instructions can be written as a flow diagram in which each item is either a description of computer behaviour, a question on computer behaviour followed by a branch, a question on user' intentions followed by a branch, or an instruction to the operator of sorts (a)to (d) above. All cards referred to must be precisely described.

Though this flow diagram may not always be the most suitable way of presenting the information to an operator it is extremely valuable in exposing wrong or imprecise formulation and should be written before the instructions are finalized.

In the final version, statements and questions referring to the computer's behaviour can be usefully be distinguished by insetting them from those referring to the operator or user.

3.2. Information relating to each section.

3.21. Specification.

The specification of each section exactly follows the pattern described for the whole program in 3.11. Computer-stored information is more usually relevant since the data and result of one section are often merely intermediate results of the whole program. It is sufficiently precise for any mistake in linking two sections together to be evident just by looking at the overall logical flow diagram and the specifications of the sections concerned. In this way the effect of a modification within a section on other sections or on the whole program can rapidly be assessed and mistakes arising from unforeseen remote effects can be avoided.

The specification states

(a) stores occupied, i.e. which DL's are overwritten with the instructions before they are obeyed.

(b) stores used, i.e. which stores (other than the ones occupied) either affect by their initial contents the outcome of the section or are themselves affected by the section. (Note that by removing and replacing the contents of a store, a program can refer to it and yet not use it.)

(c) of those stores used, which affect the useful outcome and what they contain initially.

(d) of those stores used, which are left containing information that is relevant to subsequent sections (i.e. useful outcome) and what function of the data they are left containing.

(In 3.11. it was pointed out that the information stores of the computer include TS COUNT, TCA, TCB and head positions.)

This can conveniently be given in columns with one column for the short tanks, triggers, etc. and one for each DL occupied or used, but long strings of homogeneous numbers can be more economically specified. Any store not marked or referred to in the specification is preserved by the section.

It is possible that some addresses used might only be specified parametrically, e.g. some library matrix routines use drum addresses that are calculated from other numbers found in explicitly given addresses (that state,in this case, at what address the stored matrices start.)

3.22. Logical Flow Diagram.

The logical flow diagram of a section consists of instructions and questions in English and algebra. It is followed very closely by the computer flow diagram.

3.23. Computer Flow Diagram.

The computer flow diagram consists of instructions and subroutines. It is annotated in English and algebra to show in detail the correspondence between it and the logical flow diagram. In particular each branch point is labelled with the question it asks and the possible directions labelled with the answers to this question. (The branches of discrimination instructions are also labelled + and -, or Z and NZ.)

If a subroutine is parametric the values of the parameters are written beside it. If it is modified it is marked MOD. Any subroutine must in its turn be separately documented with specification, flow diagram and coding just as for whole sections. Again, relevant information about triggers etc. is included (for example some library subroutines assume T.C.B. is off at entry and some leave it off at exit.) In the case of library subroutines used unmodified, the information in the published write-up is itself almost adequate.

Waste instructions have the form A,1-1,0,T (or A,1-1 (2) 31, T if it is also required to waste time during say multiplication.)

Destination zero instructions should be written

$$S\text{-}0_m$$

$$0_m(I)$$

where I is the address containing the basic instruction of which a possibly modified version is being obeyed from S.

3.24. Coding.

The coding specifies all instructions, numbers, subroutine parameters, punching on first four rows of triads, card numbers, and every other contribution made by the programmer to the cards of the program section. There is one column of coding for each DL occupied, whether or not it contains instructions written by the programmer. Stores occupied by a subroutine are marked with identification of that subroutine (the subroutine coding is part of the subroutine's documents.)

Occupied stores required to be clear at the start of a section are marked "zero". Occupied stores where initial contents are immaterial to the useful outcome of the section but which are affected by the section are marked "used" or possibly with some algebraic symbol.

Destination zero instructions (except possibly 17-0, 18-0) are coded 0,S-0,T,T.

3.4. Notation.

The specifications and flow diagrams described above often use symbols to identify items of information. It frequently happens that a particular symbol is used to refer to a number or instruction that varies as the program proceeds. This can give rise to no ambiguity provided that not more than one version of it exists in the computer or input cards at a time. If this condition does not hold, distinct symbols must be used, preferably prefixes and suffixes that indicate the relationship between the various items concerned. In particular this implies that any item punched on an instruction card must be distinguished from modifications of it. It is a common mistake to forget that each time a section of program is transferred to the mercury all the stores occupied by it are put back into the same state. Each reference to a number must specify the binary point since numbers (including instruction modifiers and counting numbers) may occur in versions with different numbers of binary places.

A difficulty that commonly arises is that the algebraic notation of already existing parts of the program are not in line with one another. In that case the symbolism of any section or subroutine must be defined (in writing) in terms of the symbolism of the whole program or section of which it is a part.

3.5. Precision.

One purpose of this arrangement of information is to facilitate the comprehensive checking not only of coding but also of logic that is described in 4. It will be shown there how all checking can become more or less automatic provided that the information being checked is accurate and absolutely precise. Absolute precision is quite possible and the criteria for it are:-

(a) Program specification. Could someone write a program indistinguishable from this one in respect of all input and output (except possibly for marginal accuracy) if he were given only the specification?

<u>Program section specification.</u> Could someone write a section that could replace the existing one if he were given only the section's specification?

(b) <u>Overall logical flow diagram.</u> Could someone write the specification of any section if he were given only the program specification, the overall logical flow diagram and the specification of all the other sections?

<u>Program Section Flow Diagram.</u> Could someone write the coding for the section (including punched parameters etc. if he were given only the computer flow diagram?

(c) <u>Program Pack List and Coding.</u> Could someone prepare the pack of program cards if he were given only the program pack list, the coding of all the programmer's contributions to the program, and access to a catalogued library of cards including all other routines and subroutines used by the programmer?

(d) <u>Pack Assembly Instructions.</u> Could someone prepare the card trays (including data cards and marker cards for insertions) given only the program specification relating to punched input, the pack assembly instructions, the program pack and the written data for one application of the program?

(e) <u>Operating Instructions.</u> Could someone operate the program given only the prepared card trays and the operating instructions?

If these reconstructions are possible then the collected information contains some redundancy. Just as redundancy in the numbers processed during a particular operation can be used to detect processing errors (see 2.42) so this redundancy can be used during program preparation to detect programming errors.

4. HOW TO CHECK A PROGRAM.

Programmers often say after finding a mistake by trial on the computer that they didn't or couldn't check 'that sort of thing.' Below is described a method of checking not merely that instructions are obeyed in the right order but that they do what the specification of the program says. This hinges on the documentation described in 3.

4.1. Overall logical flow diagram.

All the items of data and result listed in the specification should occur as inputs to or outputs from the overall logical flow diagram.

4.2. The Section Specifications.

The input to the whole program is the totality of inputs to each section that are not internally provided. So the punched or manual input of any section must also be described either in the program specification as data for the whole program, or else in the operating instructions as card-stored or "operator-stored" intermediate results.

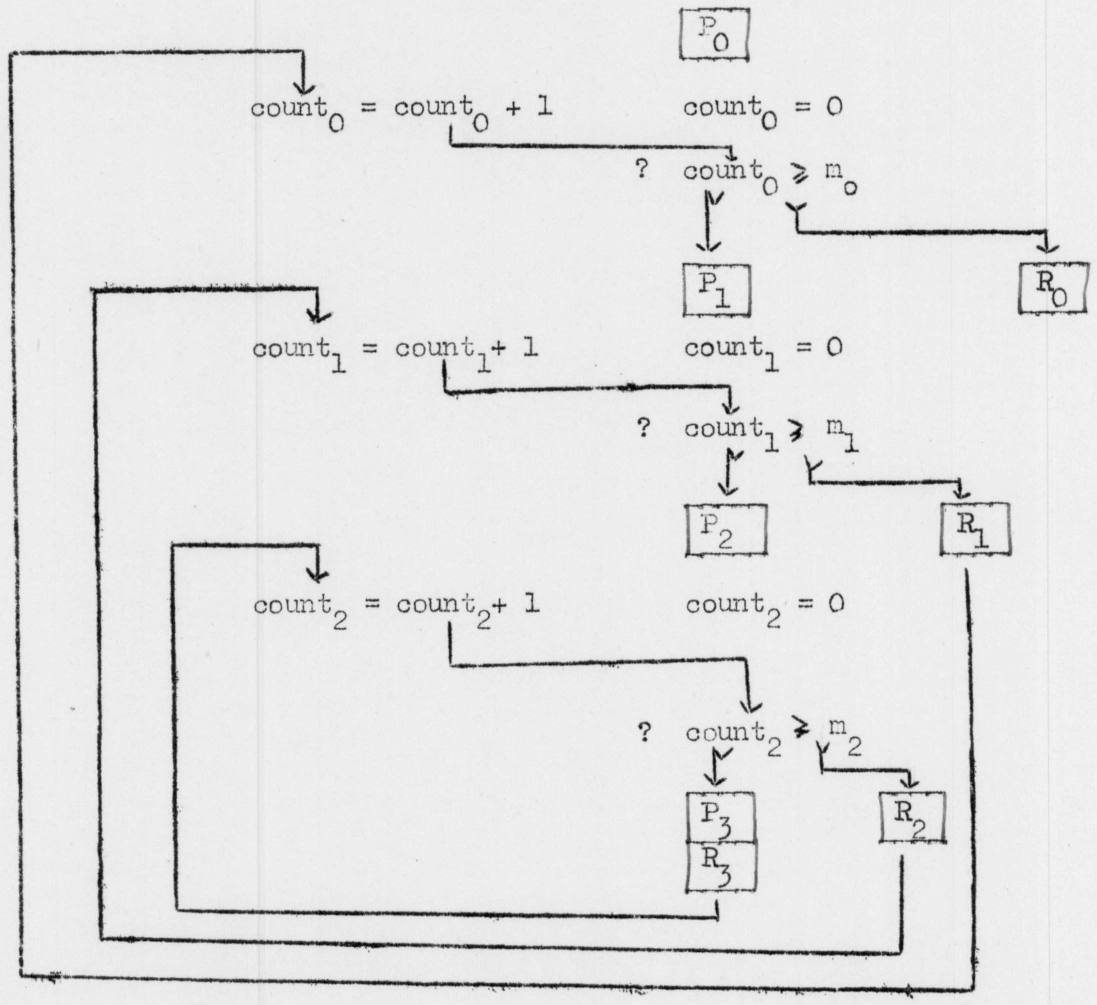
Similarly the punched or visual output of any section must also be described in either the program specification as results of the whole program or else in the operating instructions as card-stored or "operator-stored" intermediate results. A particular case of this is that any failure indication of any subroutine or routine used is also a failure indication of the whole program, although it may be possible to be more informative about its significance as a possible outcome of the program than as a possible outcome of one subroutine. (e.g. a division failure might only have one possible meaning in a particular context.) It may be necessary to add further descriptions to failure indications in order to distinguish them from one another. (Note that indications on the IS lamps given by subroutines vary according to the position of the subroutine.)

In the same way, the intermediate results provided as data for subsequent sections can be checked against one another. Any pair of sections that occur adjacently in the overall logical flow diagram should match domino-wise in their specifications (A section can have more than one immediate predecessor or successor.) If a section uses stores not used but preserved by an immediate predecessor then all possible routes connecting the relevant sections on the flow diagram must be considered.

4.3. The Section Logical Flow Diagram.

Check the logical flow diagram of each section against its specification. It should only use things that are part of its data and should leave everything that is part of its result.

Check that conditions are being correctly set for each loop especially if there is a nest of loops of the form:



There will for example be some operations, (such as modifying and replacing instructions and storing and restarting sums) that must be performed before each sequence of repetitions of the innermost loop and some that must be obeyed afterwards. These are P_2 and R_2 respectively and must be correctly composed.

4.4. Computer Flow Diagrams.

Each section of computer flow diagram must be checked against its logical flow diagram, especially the correspondence between each branch of the logical flow flow diagram and the annotations to the corresponding branch of the computer flow diagram, and also between these notes and the computer instructions used. The computer flow diagram should then be checked against the section specification to see that it does use, preserve and leave information in stores as specified. If subroutines or other blocks are used then reference to the specifications for these will be necessary, including their entry and exit points and any parameters that must be punched on their cards or provided by the program.

This specification also includes initial and final states of triggers (for example decimal punching routines require TCB to be off or they usually punch zeros.) Remember that 0-24 and 1-24 have two effects, not one, since they also clear TCB and call TCB respectively.

Check that data (including links) are set for each subroutine and inner loop from whatever direction it is approached and that the subroutine does not use stores whose preservation is assumed.

Check that each of the counting devices used (especially counting by spilling) is correct and not wrong by, say, one. Work out its effect for a simple case. Check that it is always exactly right for all uses made of it (if say it is also used for instruction modifications) especially during the first and last repetitions.

The smallest practicable units on which such checking can be performed are those whose internal organisation is tightly knit to achieve fastest possible speed or otherwise to exploit most conveniently the computer's characteristics, e.g. in subroutines, inner loops, reading and punching routines.

Then check the consistency between the annotations to the computer flow diagram and the actual computer instructions, both those associating symbols with stores and those associating a question with a discrimination instruction. Finally check that no impossible transfers are asked for and that the timing rules of multiplication, reading cards etc. are not disobeyed.

4.5. The Coding and Program Pack List.

Check the coding against the statement of stores occupied in the specification.

Check the coding against the computer flow diagram in the order in which the instructions occur on the flow diagram noting especially that branch instructions and modified instructions correctly specify their successors every time they are obeyed. Check all the coded information including entry-point, and instructions and numbers that are referred to by the program as well as the instructions that are actually obeyed.

Check the values of parameters for parametic subroutines noted in the flow diagram against the ones coded. If they do not agree then make sure that the program itself provides them. Check that subroutines used in modified forms have the modifications entered on the coding sheet and that any resulting changes in the subroutine specification (e.g. in stores occupied) is correctly recorded.

Check the program pack list against the coding (including card numbers) and against the specifications of any programs used to see that it records any tampering with the cards that might be necessary for each use. Check any parameters required by a control program against the track numbers etc. concerned.

4.6. The Pack Assembly Instructions and Operating Instructions.

Check the pack assembly instructions against the program specification and overall logical flow diagram to ensure that there are no circumstances in which the given card order would be wrong. If intermediate card storage is used check against the specifications of the sections producing and consuming these cards.

Check the operating instructions against the manual input and visual output stated in the specifications. Check that the overall logical flow diagram can be telescoped to correspond exactly with the flow diagram of operating instructions described in 3.15.

4.7. The Program Pack.

Check the cards of each section against the coding sheets by reading from the cards.

Check everything referred to on the coding sheets including initial instructions or parameters, subroutines used (identifiable by their punched catalogue number and position number), punched parameters or modifications to subroutines, F54's and card numbers. Then check the whole pack against the program pack list.

4.8. The Pack assembled for input to the computer.

Checks on data preparation prior to punching it are usually peculiar to particular jobs.

Pre-computer checking of data punching can be done with an electric card verifier provided that they are decimally punched. This is an extremely good reason for programming to accept all data in decimal. Furthermore, check-sums can sometimes be checked with a tabulator.

If a programmer is in a position to and does decide against decimally punched input (to simplify programming, or to use or be compatible with existing programs, or because the data is genuinely binary - as for example in some statistical work) he must choose a form that facilitates checking by other means. He may decide to rely entirely on programmed checks or to give the user a choice between off-computer and computer checking.

After assembling cards (and marker cards - see 3.14) in trays ready for inputting to the computer, check them against the pack assembly instructions.

Checking the data pack assembly is important before testing as well as before actual use, since mistakes may confuse the diagnosis of program faults.

4.999. Checking must be the last thing that happens before running the program either for testing or useful production. If a mistake is discovered and corrected then the revised program and/or pack must be rechecked. And so on.

5. WHAT TO DO BEFORE USING THE COMPUTER.

5.1. Prerequisites

It is time-wasting to try a program on the computer before eliminating as many mistakes as possible by systematic checking. This applies as much after the smallest alteration has been made as it does to a new program. A method of checking new programs was described in 4. ~~The special problem of checking alterations is discussed in 9.~~

It is also time-wasting to attempt unaided program testing unless you are familiar with all the facilities both engineered and programmed that make it easier. These are:-

(a) various manual inputs to the computer on the reader, punch and control panel; especially the REQUEST STOP and PROGRAM DISPLAY keys.

(b) various visual outputs from the computer on the reader, punch and control panel; especially the IS lamps and the monitors, including the use of the right hand one for inspecting TS COUNT and (indirectly) the drum.

(c) the facilities provided by the control program you are using (if you are using one) for monitoring the progress of the program, restoring control, restarting, and changing stored program.

(d) library post mortem programs, especially POST MORTEM (ZP29).

Information on (a) and (b) is contained in The Deuce Control Panel Manual and on (c) and (d) in the individual program reports (they mostly have the initials ZC and ZP respectively.)

5.2. Operating Instructions for the Program Tester.

Program testing is usually more effective if it is systematic. Just as you can ask retrospectively "How could I have prevented that mistake from getting as far as the computer", so you can ask "What could I have noticed or recorded in the behaviour of the computer that would have put me on the right track sooner". Each time you try a program on the computer you should have decided what things you are going to look for and in what order.

In 3.15 the operating instructions to the future user of the program (possibly yourself) were defined. What appears to be needed now is your operating instructions to yourself for your program testing session. This at first sight rather pointless requirement is aimed at making the thin you do to the computer as automatic as possible by doing as much as possible of the thinking before and after. The reason for this aim is that most people do not seem to find computer operation conducive to lucid thought.

These instructions will be of the four sorts listed in 3.15 but will not usually be conditional on the card output of the program (which you can take away and look at if there is any), but on information observed while it is running. So the things you particularly hope or fear might happen should be listed together with the action to be taken if they do not or do. Besides the failure indications, etc. already specified among the general documents of the program, this list will include the order and approximate extent of reading, calculating and punching, and noticable features on the IS and OPS lamps and the monitors.

Since there are not likely to be enough such landmarks to localize a divergence from correct behaviour, 5.32. describes ways in which landmarks can be temporarily introduced for testing purposes, and the sorts of information that can be collected while at the computer.

A general idea of how a program testing session can be used is given in 6.

5.3. Information Available from the Computer.

The program can be allowed to run normally, or it can be interfered with by previous tampering or manually while on the computer.

5.31. With normal running.

If a program is provided with some data and tried on the computer two sorts of information may be obtained, visual and punched.

5.31.1. Visual Information.

The lamps and monitors only have a completely determinate appearance if the computer is stopped (i.e. if the GO lamp is off). This will happen whenever the computer reaches a stopped instruction, intended or otherwise. In particular if it reaches the instruction 8,0-0 X with the reader uncalled and TS COUNT and the ID clear it is said to have "fallen out". (This instruction should not occur in a program.) When the computer is going, the general appearance of the lamps and monitors can convey useful information to an experienced operator. In particular the IS lamps should be watched while inputting the instructions and data of a program under test, since most input routines produce a characteristic pattern on them during correct running that will immediately respond to a fault in pack assembly.

A program proceeds so rapidly if allowed to that it is usually difficult to be conclusive about where it first went wrong, so 5.32.1 and 5.33.1 describe ways in which landmarks can be introduced by previously tampering with the cards or by interfering manually during operation.

However, observing and recording visual information uses a great deal of computer time and the selection from it made by an operator working under pressure rarely turns out to be the most useful, so 5.32.2 and 5.33.2 describe how fairly large amounts of it can be punched and examined at leisure.

There is nevertheless some information that can only be recorded by an operator writing it down, e.g. that cards were left unread in the reader or that they were read at 100 per minute rather than 200 per minute.

5.31.2. Punched Information.

The normal card output will be enough to show whether the program has operated correctly with the data provided. If wrong output or no output is produced it may not be sufficient by itself to trace the fault (see 7.1 for ways in which the incorrectness or absence of results can be used), so 5.32.2 and 5.33.2 describe ways of punching more evidence.

5.32. By Tampering with the Cards.

Previous remarks suggest ways in which a program can be slightly modified to provide more information, both visual and punched, about its behaviour.

5.32.1. Visual Information.

Some instructions can temporarily be made stopped by altering the program cards. These should be chosen so that as each appears on the IS lamps, some new information about what parts of the program are working is obtained. Such places are, for example, (i) just after reading program or data from the reader or from the drum (ii) just before or after entering a subroutine or a new section of program such as a repeated loop. The computer stops every time it reaches a stopped instruction until it is given a manual or automatic oneshot. This fact has two consequences.

(a) a temporary stopper may be a nuisance if there are many repetitions of it before the program reaches a point to be investigated by the test.

(b) a temporary stopper will not stop the computer just before or in the middle of a reading or punching routine unless special precautions are taken. It will merely affect the result of the routine by taking up a oneshot intended for some other stopped instruction. Ways of examining such routines in action are described in 5.33.

```
Thus              st  Read
                   30-1_0X
              1_0
```

will call in a program 2mc out of phase. Such effects may be more drastic and obvious or may be partly masked by later parts of the program being wrong in the same way.

(c) a temporary stopper can also affect the subsequent outcome if it prevents synchronisation between multiplication (or division) and instructions obeyed while it is still going on.

5.32.2. Punched Information.

Some sort of "progress report" can be programmed that periodically indicates on the OIS lights or by punching a card how far the program has got, with perhaps some other useful information. This bit or program can be short-circuited once it has served its purpose. (However, after doing this further checking and testing must be done.)

5.33. By Manual Interference.

The usefulness of a trial run can sometimes be further improved by manually interfering with it. This should be planned in advance to produce the visual and punched diagnostic evidence that will detect and locate a fault.

5.33.1. Visual Information.

The "landmarks" inserted previously can be supplemented by manually slowing down the progress of the program so that significant features on IS lamps and monitors can be observed. There are several ways of doing this.

5.33.11. Request Stop.

Request stops differ from temporary stoppers as follows:-

(a) Any selection of specific instructions can be made temporary stoppers and the computer will stop before every attempt to obey one of them until it gets an automatic or manual oneshot. A request stop on the other hand can not be made on a specific instruction but only on a pattern comprising one or more of NIS, Source and Destination, and any instruction conforming to the pattern will stop the computer. Furthermore, only one pattern can be specified at a time, though this might cause a stop at many instructions (possibly unexpected ones.)

(b) A request stop will freeze the computer during a reading or punching routine with any card movement continuing indefinitely without affecting it (consequently the subsequent outcome after passing the request stop will be affected.)

(c) A request stop on 0-24, 1-24, 2-24, 10-24 or 12-24, 17-0 or 18-0 can affect the subsequent outcome, and consequently has very limited use.

(d) A request stop can only be introduced or removed while operating the computer and so can for example be removed after a few repetitions of a suspect loop.

(e) A request stop can be made on an instruction whose address is not known, for example to detect unintentional transfers to a particular tank. Concomitantly it may confuse the operator by stopping on unsuspected instructions that are like the instruction that is intended, unless care is used in previously selecting the instruction.

5.33.12. Have the computer at STOP, possibly with CONTINUOUS SINGLE SHOTS. This produces a static or slow motion picture of the lamps, etc. However, remember that it does not distinguish between stop instructions and go instructions.

5.33.13. Have the computer at AUGMENTED STOP with CONTINUOUS SINGLE SHOTS (or PROGRAM DISPLAY which is quicker and in every way preferable.) This has the same effect as the above but stops on stop instructions.

5.33.14. Prevent the normal flow of cards through the reader with the reader STOP key or the SINGLE READ key or by splitting the input pack. This may be a strategic moment to observe the IS lamps or monitors. For example the reading of a card can be simulated by setting on the ID lamps the contents of consecutive rows and giving manual oneshots.

5.33.15. Prevent the normal flow of cards through the punch with the punch STOP key. For example the punching of a card can be simulated by giving manual oneshots and observing (and clearing) the successive contents of the OPS lamps.

5.33.2. Punched Information.

The following information can be punched by interfering manually:-

(a) the sequence of instructions actually obeyed (punched by PROGRAM DISPLAY, an engineered device.)

(b) the actual state of the computer at any one time (punched by POST MORTEM, a programmed device.)

5.34.

It must be remembered that altering the operating circumstances to produce useful evidence may not produce evidence about normal operation. For example, instructions following multiplication or division might have different outcomes when obeyed stopped, and PROGRAM DISPLAY might supply a TIL signal when its absence is expected in normal running.

5.4. Test Data.

In deciding what data to use at each test run the object is twofold:-

(a) to subject the program to slightly more severe testing than it is known to stand up to, so that one new feature is being tested. If possible use data such that hand checking of the calculations is feasible if it becomes necessary.

(b) to cover in as short a time as possible all the limiting cases of the data, testing each parameter to the limits prescribed for it by the program specification.

For example, the first run should be done with as few and as simple numbers as will check that the overall logic is being followed. Thereafter, if the program works with some data and not with other, cases that will isolate characteristics that might be relevant should be constructed.

5.999.

Checking must be the last thing that happens before testing the program. If any alterations are made during preparations for testing, then the revised program and/or pack must be rechecked. And so on.

6. WHAT TO DO WHILE OPERATING THE COMPUTER.

Normally a trial run can be conducted according to operating instructions previously prepared. However, these must allow for a completely unforeseen eventuality.

6.1. Emergency Drill.

If something happens that has not been specifically planned for, restart the program, program display from the last correctly occurring landmark up to the point when something has gone noticeably wrong and then post mortem. If there is no time to restart, program display a few instructions if the computer is still going and post mortem in any case.

A complete post mortem may take up to eight minutes, but some of this time can be saved if it is known that only certain areas of the drum store are likely to contain useful evidence. The mercury store should always be punched complete in these circumstances.

6.2. Further detail in dealing with unforeseen circumstances.

There follows more detail of a possible procedure which can be ignored or varied in the light of experience.

Watch the IS lights while the program is being read in. They may show that the input is not happening properly. If so, by splitting the pack find where it is going wrong. (see 5.31.1 for further remarks on this). When the machine unexpectedly appears inactive, i.e. when neither the intended thing nor anything in particular is happening, then either,

(a) the GO lamp is on and it is going, or

(b) the GO lamp is off and it has reached a stopped instruction indicated on the IS lights.

In case (a) the machine may be in a fairly small loop.

Examination of the TS's and DL's with the monitor may suggest:

(i) that nothing is apparently changing

(ii) that something is changing with periodicity

(iii) that something is changing progressively, e.g. a counting number is counting a very large number of iterations.

In case (iii) note where the progressive change is. In any case use PROGRAM DISPLAY and while this is punching continue to watch the monitors and also the IS lamps again for any observable periodicity. If the change is completely periodic program display need be run for one period only to obtain full information. However, whether this is so is not always obvious from watching the control panel and when the punched evidence is examined larger periodicity than the smallest one might be discovered.

After stopping program display, use POST MORTEM to recover as much of the machine as might be relevant.

In case (b) see if the reader or punch has been called. If both have, there is probably something wrong (e.g. the reader should have been cleared.) If one has and is not ready, then the stopper is probably the one that is waiting for the first row of a card. In the case of the reader, run in the cards that the computer is expecting. If you do not know what it is expecting there is probably something wrong. However, it may be that the reader has been called, or merely not cleared, by mistake. In the case that the punch has been called either there are cards in the punch hopper or it has run out of cards. In either case run in cards. If the stopper is not explicable in this way, then it is

(i) a failure or other indication in a program or subroutine that you are using. In this case do as you have previously planned;

(ii) a temporary stopper that you have supplied in which case you will already have decided what action you intend to take on reaching it;

(iii) a stopper you did not expect (you may recognise or remember that it is an instruction you did not intend to be a stopper in which case a oneshot will be sufficient, or if it is very frequently obeyed it might be necessary to change it on the cards and read the program in again.);

(iv) an instruction you did not expect, e.g. you might be quite certain that your program does not contain an instruction with NIS 7 say. Look at TS COUNT with the right hand monitor in order to see the P1 and P15 digits (which are not shown on the IS 1S lamps.) You might now decide that a word that is not an instruction is mistakenly in TS COUNT. Record as much of TS COUNT as is possible. You can now either

  (a) use POST MORTEM and discover at leisure where the spurious instruction was, or

  (b) try to identify it in the monitor and then by stopping the machine and one-shotting try to re-enter the program albeit at random on the chance of getting more information;

(v) probably the commonest unexpected thing to happen is that the IS lamps (including the GO lamp) are blank. The computer is said to have fallen out. Proceed as in (iv).

6.3. Making the Best of a Bad Job.

After finding a symptom at the computer it is sometimes possible to force the program to proceed, though incorrectly, and collect more information about the first detected, or other, mistakes. It may be useful to alter the course of the program by manual interference. For example,

(a) a branch instruction can be forced the wrong way with the DISCRIM key;

(b) the contents of a tank can be altered with the EXTERNAL TREE key and the CONT TT key;

(c) the NIS, Source or Destination of an instruction can be altered as it is being obeyed with the EXTERNAL TREE key and the SINGLE SHOT key.

The computer must be stopped to do any of these.

6.4. Inconsistent Behaviour.

Apparently inconsistent behaviour can have several causes, depending partly on how loosely you use the word. Inconsistent behaviour can only be confidently deduced from the evidence if identical operations that fulfil the rules for using the computer started from states of the computer that were identical in all relevant respects and led to different results.

Apparent inconsistencies may be caused by:

(a) different initial states, e.g. any of the following might be relevant; (i) the contents of the drum were different. (This can usually be avoided by using CLEAR DRUM ZP 13/1 after the initial card of a program); (ii) the drum head positions were different (CLEAR DRUM ZP 13/1 also leaves these in a constant state); (iii) the punch was in different state (RUN IN on one occasion and not on the other). For complete information about the correct initial state see 2.11.

(b) using forbidden sequences of instructions or indeterminate operating instructions. These are (i) sequences depending on Hollerith times outside the prescribed limits (especially the time after reading program cards during which it is not possible to read the ID lamps); (ii) sequences depending on magnetic drum times (in relation to Hollerith times) that are outside the prescribed limits; (iii) unlisted instructions with destination 24. (iv) Automatic counting in QS 17 or 18 that spills from one word to the next.

(c) different operations being compared, e.g. (i) two runs during one of which temporary stoppers, or controlled passage of cards or program display was used may produce different results if the time taken by cards to go through the reader or punch, or by the multiplier or divider is relevant; (ii) the use of PROGRAM DISPLAY may interfere with a program using TIL; (iii) the INITIAL INPUT and RUN IN keys are only interchangeable in certain contexts.

7. WHAT TO DO AFTER USING THE COMPUTER.

7.1. The use of incorrect answers.

The cards punched by the program being tested supply information of two kinds:

(a) the number and general layout of the cards, e.g. a systematic displacement of the rows of cards can be caused (i) by having extra stoppers in the punching routine that are being taken up by oneshots from the punch, (ii) by instructions which should be stoppers not being stoppers. These two mistakes tend to cause a spreading out of rows and a telescoping of rows respectively.

(b) the actual numbers. Most that could be said under this heading must relate to particular calculations, but some general observations are possible. Frequently the fact that numbers which should be the same are in fact different or the reverse, is sufficient to point to a cause. For instance, a string of numbers that should be equal, having been produced by repetitions of a loop on fixed data, might in fact have the first one different from the others. This suggests that the conditions for the loop to work are only being set correctly after the first time round it or are being destroyed after the first time round it.

7.2. The use of PROGRAM DISPLAY results.

PROGRAM DISPLAY cards can be checked against the flow diagram of either your program or the subroutines, etc. that you use.

In the case that they display the machine falling out or reaching an improbable instruction, it is best to look right away at the last card punched and trace back to the incorrect instruction.

In the case that an unknown section of program has been punched it can best be identified by finding distinctive instructions (e.g. a wait number of 31) on the coding sheets. Note that

(a) the NIS of each instruction specifies the DL in which its successor is stored;

(b) there may be more than one set of instructions stored in a particular delay line at different times;

(c) the instructions in question might be from a subroutine and not on your coding sheets.

A large number of cards can be rapidly examined if a periodicity of say 25 instructions is present or if an instruction is repeatedly obeyed with modified wait numbers, or by merely looking for successive branch instructions and checking that the right direction is taken each time. An error can thus be rapidly located to a few dozen instructions.

An incorrect sequence of instructions may be explained by disagreement between flow diagram and coding or between coding and cards or both. If it is not, the first incorrect instruction that way obeyed has somehow wrongly occupied an address. Establish whether or not the sequence of instructions had been obeyed correctly before. Either the PROGRAM DISPLAY cards or other evidence about how much of the program must have previously been completed may do this. If it has, then the incorrect instruction has been placed there since the last time. The program display cards may actually show this happening.

Alternatively discover what the incorrect instruction is. It may be an instruction or a number which should be somewhere else. If so, see whether it is in its right place as well and if not what is. POST MORTEM results will be necessary to do this.

7.3. The use of POST MORTEM results.

POST MORTEM might have been used with a specific purpose, e.g. discovering whether certain data were stored in the correct places, whether certain numbers checked against each other or comparing the entire store at the end of some section with the written specification. When something unexpected has been found further use can sometimes be made of them to explain it.

In this case or if they have been produced without a specific intention the following things can be examined:

(a) Counting numbers or programmed triggers.

(b) Modified instructions.

(c) Addresses to which instructions are sent during the program, especially 1_{30}, 1_{31}, etc. if subroutines are used (though some subroutines use non-standard link positions.)

(d) Addresses to which numbers are moved consecutively during the program, e.g. a DL which holds successive tracks of the drum, or a TS which holds successive words of a DL during the program.

(e) Sums cumulated during calculations.

(f) Addresses to which newly generated numbers are sent.

Some of these things may indicate how far the program has progressed. All of them should be carefully examined since apparent contradictions between them may shed light on the error.

Note that programs of subroutines that you do not write might leave unfamiliar stuff around.

7.4.

After finding a mistake as a result of evidence collected from the computer, you should make sure just what the mistake accounts for. It may be that the evidence already available will lead to further mistakes being found.

8. ERRORS OTHER THAN PROGRAMMING ERRORS.

Running a program may reveal programming mistakes, i.e. inconsistencies, ambiguities or omissions in the documents listed in 3. (Any programming mistake could have been discovered without resource to the computer.) It may also reveal non-programming mistakes of two sorts:- unexpected or unplanned features that call for modifications in the program, and mistakes in preparing for, operating and subsequently processing the results of, the current run.

During any investigation of a program these two additional possibilities should always be borne in mind, in order to avoid a fruitless search for a programming mistake to account for what is inherent in the design of program or peculiar to one run.

8.1. Pre-Programming Errors.

There are three sorts of pre-programming mistakes that cannot be revealed by checking a program off the computer:-

(a) a discrepancy between the specification and what was intended. This usually arises as a misunderstanding between the programmer and the commissioning person or people and considerable effort should be put into avoiding this by insisting on precise and comprehensible exchanges of information at an early stage. This can be very irritating to a commissioner. It is a phase of programming when redundancy should be piled on, especially by quoting simple and extreme examples. Many commissioners will agree (perhaps with some irritation) to four synonymous formulations of their problem and then demur at a fifth. In particular an unsuspected absurdity is often revealed if they are made to visualise what numbers they will actually have to provide or what their results will look like in print.

(b) an unsatisfactory implication of the specification that is only revealed by trying the program. For example, an iterative procedure may not converge rapidly enough or the calculation may require greater accuracy or the necessary operating may prove impracticably difficult or delaying. This usually arises in cases where the numerical, statistical or arithmetical analysis to investigate the required accuracy or amount of computation or the operating implications was too difficult or was not undertaken because not considered.

(c) a mistake (possibly an ambiguity or omission) in the information about a part of the program previously written (by the programmer or someone else.)

8.2. Post-programming Errors.

There are three kinds of post-programming errors that may occur during a particular run. However, they may not be immediatly distinguishable by their symptoms from other sorts of mistakes. Furthermore, though they can be corrected by re-running the program they may suggest program modifications and so fall into type (b) of 8.1.

(a) A mistake in preparing the data according to the program specification or in assembling the pack according to the pack assembly instructions, i.e. cards in the wrong order or with wrongly punched parameters. If the program has a specified result for any data, correct or incorrect, then the latter will be immediately identified. The usefulness of this is somewhat reduced by the fact that the second sort of mistake or any other input without a specified outcome may produce a misleading outcome.

(b) An operating mistake. The opportunities a programmer has for encouraging these have already been mentioned.

(c) A mistake in the subsequent processes. After the computer has produced the right cards, sorting, printing or copying errors may occur. The actual cards produced by the computer with sufficient identification for their original order to be reconstructed should be available while investigating a suspected fault.