Programming procedures

ICL

Programming procedures

ICL

ICL International Computers Limited

User notice

MANUAL (NOTICE NO.)

22/5/70

4202 PROGRAMMING PROCEDURES (1)

File one copy of this notice with each of the manuals indicated.

PROGRAMMING PROCEDURES (1ST EDITION)

The following errors in this manual should be corrected:

Page 6	Line 21	X has been omitted between the two parts of the diagram.
Page 6	Line -11	"Suite" should read "project".
Page 106	Line 24	Insert after "This does not exist" the following: "on 1900 files or".
Page 135	Line 22	"$^4/_5$ x effort" etc. should read "4 or 5 x effort" etc.
Page 188		The heading "Use" should read "Flowchart ZC" not "Flowchart DC".
Page 241	1st example	Delete COMP. from first line and replace with a full stop. On the other four lines insert COMP before SYNC RIGHT, leaving one space character before and after.

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1JP

FORM 1/2304/45(2.69)

Programming procedures

A professional approach to the planning and control of programming

ICL

Programming procedures

A professional approach to the planning and control of programming

ICL

A

Technical Publication 4202

First Edition April 1970

Issued by Technical Publications Service
International Computers Limited
Head Office: ICL House, Putney, London SW15
Produced by ICL Printing Services
at Letchworth, Hertfordshire

Preface

This manual gives practical guidance to programming staff on the reasons for and requirements of their tasks, the factors they should be aware of in making intelligent decisions in the course of their work, and above all presents reasoned programming standards in those areas which are crucial to good programming practice and management.

It is primarily intended for those more experienced programmers who are directly concerned with day to day development and supervision of projects and who have a responsibility of involvement with planning.

It is essential however that management within the data processing function should fully understand the bulk of the material presented and make the final decision on how they intend to implement it.

As with its companion, *Systems Procedures* (Edition 1, TP4196), some parts of this manual develop the appreciation of the subject given in *Techniques of Computer Management* (Edition 1, TP3357). However it is designed to stand on its own for the benefit of a different readership and therefore has its own scope and arrangement of material.

Contents

PART 1 GENERAL

Chapter 1 Introduction

THE ROLE OF THE PROGRAMMER

Programming constitutes the detailed planning and implementation of the computer aspects of a data processing system. As such it is preceded by the systems design functions which prescribe the requirements to be fulfilled by the programming function. The main objective of programming is therefore seen as the production of programs which can be proved to meet the systems requirement. Thus the function has a clearly defined end point although the interface between what is regarded as the beginning of the programming function and the end of the systems functions may vary according to the particular organization.

The size and complexity of the programming task naturally varies according to the nature of the data processing system but in every case the task involves a sequential series of special sub-tasks ranging through planning, coding and testing. In carrying out each of these sub-tasks, certain guidelines and principles of good practice have been evolved over the years of ICL's experience and these should help to ensure that each sub-task can be carried out most efficiently and effectively, particularly as regards the performance of the resulting computer system.

THE OBJECTIVE OF THE MANUAL

The main objective of this manual is to make available to senior programmers guidance on good programming principles, and to provide methods for efficient programming practice. However a substantial part of the guidance is concerned with detailed programming and this will also be valuable to less senior programmers.

Since the manual is intended to be applicable irrespective of the type of ICL processor it is inevitably not comprehensive in those detailed matters such as software, which are particularly machine dependant. Nonetheless it is felt that the manual will provide most of the guidelines needed within a reference work of good programming practice.

The organization, scheduling and control of a programming project is given special consideration in the manual. The aim here is to provide guidelines to minimize the risk of late completion dates and of unmaintainable but elegantly written programs. Bitter experience has taught many people that detailed scheduling and control is essential, particularly for larger projects, but for many managers it remains an extremely difficult task. This arises because it is difficult to gauge the real progress of work done up to an interim point. In particular it is very difficult to judge the quality and soundness of work done until it is well under way. Management may not have complete confidence in a work schedule unless they themselves have been involved in the detailed planning. This is usually not practicable and in any event a more feasible solution is to institute a standard procedure of planning and control which is clear to the programming management and which thus gives them confidence that their staff do plan and control in a systematic and thorough way. The standard approach should also bring about an improved efficiency in co-ordinating the efforts of a programming team and, by removing some of the mystery from plannning and control, should help in the development of potential senior staff. The procedures described in this manual are designed to facilitate the setting up of an organized system of control for programming projects and this second objective of the manual is aimed at the programming managers, chief programmers and all senior programmers who have responsibility for scheduling and controlling the work of other programmers.

While users may not use the precise procedures set down in the manual, it is strongly advocated that these, or similar procedures, should be used to meet the underlying principles of disciplined planning and control. Use of such procedures will lead to the provision of well documented and easily maintainable systems and ensure that the most efficient use is made of resources available.

While the main objective is to provide guidance for senior staff, it is felt that more junior staff should benefit by acquiring an appreciation of the principles of good programming organization and control as well as the detailed programming tasks.

THEME OF THE MANUAL

The manual therefore aims to give guidance on two main aspects:

1 Programming practice

2 Organization and control of resources

The manual is organized to present in chronological sequence the guidance or practice in the different sub-tasks within programming. The organizational aspects are introduced in the manual, interspersed between relevant programming sub-tasks where it is felt that consideration of these aspects would mainly occur.

As it has already been stated, the beginning of the programming task and the end of the system task varies according to interpretation by different users. Differences are mainly concerned with the degree of expansion of the systems design to show the organization for the computer. In order to allow for the widest interpretation of the programming task a description of this detailed design of the computer system is included in the manual and this precedes the section on program planning, that is, the design of individual programs which may more generally be regarded as the beginning of the programming task. The ICL manual *Systems Procedures* (Edition 1, TP4196) will give a further understanding of the systems function and its relationship with the programming role. Whatever the formal dividing point between responsibilities, it is vital that there is close liaison so that the programmers viewpoint is taken into account particularly when designing the computer orientated aspects of the system.

CONTENT OF THE MANUAL

The manual is in three parts which deal in turn with the basic principles of organizing programming projects, planning considerations and program writing, acceptance and maintenance. The sequence of chapters and sections approximately follows the sequence of activities. Each chapter is concerned with a different level of programming and any detailed information is given in an appendix at the end of the manual. It is intended that the headings give a framework that could be used for a user's manual of programming standards.

Part 1 General

CHAPTER 2 PROJECT ORGANIZATION

This chapter considers the advantages of dividing project work into stages, the possible ways of allocating responsibilities, the necessary standard documentation and the need for controlling implementation.

Part 2 Planning

CHAPTER 3 PROJECT DESIGN

This chapter covers project design from the programming viewpoint. Techniques are given for designing suites of programs and the connecting files. Guidance is provided on the checking of the systems specification and the presentation of the suite design and requirements.

CHAPTER 4 IMPLEMENTATION METHODS

This chapter outlines the overall considerations which must be taken into account in forming the main strategies for the implementation of a project.

CHAPTER 5 PROGRAM DESIGN

Factors which must be considered when designing programs are described. Techniques for various aspects of design are given and standards for program specifications are included.

CHAPTER 6 RESOURCE PLANNING AND CONTROL

Methods are given for estimating resource requirements, planning the use of these resources and the control of actual performance.

Part 3 Program writing, acceptance and maintenace

CHAPTER 7 DETAILED PROGRAMMING

This chapter sets out in detail the way in which programs can be produced effectively, and includes recommended standards and techniques for all detailed programming tasks.

CHAPTER 8 PROJECT ACCEPTANCE

The tasks involved in proving the acceptability of the complete project are discussed, and guidance is given on the documentation required for efficient operational running.

CHAPTER 9 MAINTENANCE

Methods are described by which program amendments, necessary during operational running, can be carried out.

Appendices

There are fifteen appendices which give detailed information on the subjects included in the main part of the manual.

The separation of discussion and detail make it possible for users to adapt the manual either as a guide or as a desk handbook of installation practice, and to add material to the framework to meet local needs.

Chapter 2 Project organization

This chapter covers the basic principles on which the manual is based. It includes an outline of project stages, allocation of responsibilities, areas where standardized documentation can be introduced and also areas where controls are necessary. These four subjects are closely connected since the project stages to some extent define the responsibilities to be allocated, and the areas of standards and controls are dependent on the project stages as well as being inter-related.

PROJECT STAGES

The implementation of a complete project, that is from the initial concept to live running, covers in most cases a long time span and requires a considerable amount of man-effort, and it is essential that the time span is controlled and the effort used effectively if the project is to be brought to a successful conclusion.

In order to introduce the concept of control, and to provide a basis for a standard method of implementation, it is necessary to break a project into stages each of which is more easily manageable than the project as a whole.

A project stage can be defined as a relatively self-contained activity which has a definite end point. This point should be reached before the next stage is commenced. The breakdown into stages is obviously essential in large installations where different people are responsible for the various project activities, but even in small installations, where only one or two people are concerned, it is equally essential if projects are to be carried out with any degree of order.

Stages

The stages into which projects are divided, bearing in mind the definition of a stage, are as follows:

Systems study

Systems specifying

Project design

Program design

Detailed programming

Project acceptance

Maintenance

The detailed activities and end points of most of these stages are described in the rest of this manual. The only stages omitted are those of systems study and systems specifying which are considered to be activities in which a programming supervisor plays no part.

Advantages

There are several benefits to be gained by breaking a project into self-contained activities:

1 Staff are disciplined to concentrate on finding, and presenting solutions to problems, at the appropriate level rather than becoming immersed in a mass of unco-ordinated detail.

2 Staff can be used more efficiently since the activities can be allocated according to experience and ability.

3 The production of documents is enforced since these act as the communications between stages. These documents can be checked by the recipients thus ensuring that there is agreement on the contents and also detecting errors before too much effort has been expended.

4 Documents can be produced according to standards, which aids understanding and maintenance.

5 The completion of each stage forms a natural check point for control purposes.

Any attempt to implement a project without this discipline of activities and end points is likely to result in a sub-standard product which neither satisfies the user's requirements nor has sufficient documentation for maintenance purposes. It is also highly unlikely that any target dates will be achieved.

ALLOCATION OF RESPONSIBILITIES

There are many different ways of allocating the various activities to computer personnel. These can vary from one person carrying out the complete range of activities, to a very detailed breakdown of sub-tasks amongst staff in systems and programming departments.

This manual is intended for the use of any installation and the variables of number of staff, background, training and experience, size, complexity and technical novelty of application make it impossible to use job titles (senior programmer, assistant programmer, etc.) to clearly indicate responsibilities. This section of the manual therefore concentrates on the responsibilities implied by different functions which are of greater or smaller importance in any project. The stress is mainly on the planning and supervisory functions, but implied and included in these are the more detailed functions which are the subject of supervision.

The table below illustrates four possible methods of division of responsibility (but does not claim to show all cases) relating function to personnel category. The point marked X shows the division between business-oriented work and computer-oriented work.

Stage	*Principal result*	*Possible division of responsibility*			
Systems study	Statement of requirements	Systems analyst/ programmer	Systems analyst	Systems analyst	Systems analyst
System specifying	System specification				
Project design	Outline suite specification				
Program design	Program specification			Programmer	Programmer
Detailed programming	Working programs		Coder		
Project acceptance	Validated suite and system		Systems analyst / Coder	Systems analyst / Programmer	Systems analyst / Programmer
Maintenance	Amended or corrected system				

X

Accepting that the earliest likely programming stage is the suite design, proceeding from the systems specification, this manual covers aspects of the work from this point on.

Each section includes discussion of appropriate allocation of functions to staff. A number of factors are essential in all decisions on allocation and it should be a management standard (which is not within the scope of this manual) to ensure that they are borne in mind. These factors are that:

Each member of staff should be made aware of his functions and responsibilities.

His experience should include formal advanced training to anticipate extending his responsibilities.

He should receive adequate additional supervision when faced with new responsibilities, and time scales should reflect the training and supervision effort required.

Functions

The following is a list of the responsibility functions grouped under the project stage headings:

SYSTEMS STUDY

Collection of all facts concerning the requirements of the user system and production and agreement of the statement of requirements

SYSTEM SPECIFYING

Production of a systems specification

Obtaining agreement on the detailed requirements of the system

PROJECT DESIGN

Design of a computer oriented system and production of a document describing the computer system

Choice of method of implementation and the program languages to be used

Obtaining agreement to the suite design

Initial estimating of resources required and time scale

PROGRAM DESIGN

Planning and structuring each program and specifying the requirements of each program

Estimating the effort required and planning the use of resources

Obtaining agreement on the contents of the program specifications

DETAILED PROGRAMMING

Control of implementation and standard of work

Supervision and training of junior staff

Checking of flowcharts and coding or delegation of these tasks

Production of working programs, and operating instructions

PROJECT ACCEPTANCE

Testing of the suite to the satisfaction of all users

Production of a comprehensive operating procedure guide, and on the job training of the operating organization in the running of the project

Production of a comprehensive guide for user departments and on the job training of these departments in the use of the system

Assisting user departments to make best use of the system and ensuring that results are satisfactory

Collection of all project and program documentation to form a project file for future reference

MAINTENANCE

Correction of program errors

Assessment of new requirements and implementation of these when agreed

Updating of documentation

Advantages

The allocation of responsibilities enables installations to develop specific job definitions within their departmental structure. These job definitions can then be used to develop firm lines of staff reporting and progression. In a situation of staff shortage and high staff turnover it is most important that staff can see what their current job entails and that there are possibilities for progression.

From the point of view of orderly implementation it is essential that an individual is made responsible for each function and that the demarcation lines between functions are clearly defined. If this is not done functions will be omitted or skimped, or there will be continual overlap and duplication of effort.

There are also advantages in allocating the main responsibilities to different individuals since this introduces automatic checking of the end products of each stage.

STANDARD DOCUMENTATION

Standard documentation covers two areas; the documents which are considered as standard documentation for any project, that is their existence is essential, and the standards for the production of each of these documents.

Documents which must be produced, regardless of installation or project size, are those which form communications links, either for agreement purposes, or between stages, or for maintenance. The production standards vary according to the type of document and range from content headings for specifications to detailed format and referencing standards for source coding.

It is essential that standard documents, once produced, are kept updated by any changes, and that these changes are made consistently to all the relevant documents. To this end all installations should set up a system by which amendments are listed, together with the date of amendment and the documents affected. This list should be circulated to all concerned so that they can ensure that their part of the documentation is updated.

Documents

The basic purpose of the documents which must be produced are described briefly below. Further details of the documents of concern to programming are given in the rest of the manual.

STATEMENT OF REQUIREMENTS

This gives details of what the user requires of the system.

SYSTEMS SPECIFICATION

This is an extension of the statement of requirements and gives all detailed information needed for designing the suite. It forms the basis of agreement between all departments concerned with the project. The contents may be extended to include a specification for the suite design and may include broad estimates of time scale.

OUTLINE SUITE SPECIFICATION

This is only produced if the systems specification does not contain the specification for the suite design. It is intended as a means of reorienting a general job description to a computer design document, and forms the communications link between the suite designer and the program planner.

PROGRAM SPECIFICATIONS

These are produced for each program in the suite and contain all the information required for writing the program. They form the link between the program planner and the programmer as well as being vital for maintenance purposes.

FLOWCHARTS

Flowcharts are produced for each program in order to establish the program structure and the logic. They are used as a basis for coding and also for maintenance.

SOURCE CODING

Coding obviously has to be produced as a communication with the computer but it also required for maintenance.

OPERATING INSTRUCTIONS

These are produced for each program and contain details of how the program is to be run on the computer. They form the link between the programmer and the operator.

OPERATING PROCEDURE GUIDE

This is produced for a project and is the communication link between the project development team and the operating organization. It contains full details for data preparation, job assembly and result despatch, and includes the individual program operating instructions.

USER PROCEDURE GUIDE

This is produced for use by the departments who supply the data and/or receive results, and contains details of how forms are filled in, batch controls, etc.

Advantages

The production of standard documentation allows the establishment of control since the documents are the end product of a stage, and the stage can be considered as completed once the relevant documents for that stage have been checked by the recipients.

The use of standard contents eases the problems of communication since a document in a known format is more readily comprehensible, and also ensures that important subjects and facts are not omitted. Additionally, standardized documents are more easily referenced and updated, and so make maintenance easier and increase the reliability of the contents.

The penalty for not producing standard documentation is twofold. Firstly there is no guarantee that the suite fulfils the requirements since no checking will have been possible. Secondly the suite cannot be maintained since it will not be possible to assess the effects of changes, or to include corrections or changes in the programs.

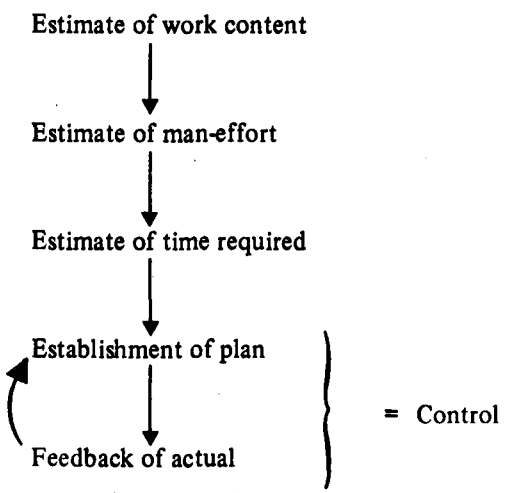
CONTROL OF IMPLEMENTATION

There are two aspects of implementation control. One is the day to day supervision of staff, not only to ensure that they are present and working, but also to ensure that their work is accurate and produced according to the required standards, and to provide adequate training.

The other aspect of control is the measurement of progress against a predetermined plan so that deviations from the plan can be seen and corrective action taken. The extent of progress control is likely to vary according to the size of installation, however all installations need a basic method in order to use man-power effectively and to set and achieve target dates.

Progress control

In order to control there must be a plan against which to measure progress, and in order to establish a plan there must be an estimate of the effort and time involved. The basic structure leading to control is therefore:



The effectiveness of the control relies on the accuracy of the original estimates and it is important, therefore, that the work content estimates are based on factors which can be analyzed objectively in the case of differences between actual and estimated performance, and which can also be used to improve estimates for future projects.

Each project stage provides an area to which this basic structure can be applied, and the standardized contents of documents facilitate the setting of standards for estimating work content.

The stage which is of particular concern to programming supervisors is that of detailed programming, which can be excessively time consuming if uncontrolled. Since this stage is very critical it is necessary to apply stringent controls to both the writing and trial activities. Detailed methods of control are described in Part 2, Chapter 6.

Advantages

The controlled implementation of a project means that target dates can be achieved without a situation of continual crisis which inevitably leads to a sub-standard product, and that other departments concerned with the project can be given a chance to prepare for the changeover to the computer system. Additionally the use of resources is planned and best use can therefore be made of staff. If inaccurate estimates or changes in requirements make alterations of plan necessary, the affect of these changes can be seen at an early enough stage to allow for considered corrective action.

The provision of adequate supervision, although it may appear a costly overhead, avoids wastage of effort, ensures an acceptable end product and allows for on the job training.

Without supervision and control it is impossible to assess the progress that has been made on a project and it will not be realized until too late that the work is incorrect or that the target date will not be achieved. This of course leads to panic measures to finish the project on time which in turn means that stages are omitted thus increasing the maintenance required, or that staff are taken from another project which will then itself become critical.

PART 2 PLANNING

Chapter 3 Project design

It has been mentioned in Part 1, Chapter 2, that the responsibility for the computer system design of a project varies according to installation. In some cases it is considered as a systems function, in others a programming function, and a further possibility is that it is treated as a combined function according to the ability of the staff.

Whichever method is used it is necessary for any experienced programmer concerned with the development of projects to be aware of the factors which must be considered and the documentation necessary. They will then be capable of either carrying out the suite design function or of checking that the design produced by systems people is complete and adequately satisfies the requirements.

This chapter first describes the final documentation of the project design stage, and then gives details of the factors which must be taken into account either when designing or when checking the design. Guidance is also given on the information which should be available, and which should be checked, before any suite design can be started, namely the requirements of the overall system.

THE OUTLINE SUITE SPECIFICATION

The outline suite specification is the document which describes the computer oriented system at a broad level. In some installations such a description is considered to be part of the system specification; however there are other installations where the systems specification excludes the suite design, and it is therefore necessary to produce a separate document for the computer designed system.

Whichever method of presentation is used and regardless of installation size the description of the computer system must be considered as essential documentation since it is the end product of the suite design stage and forms the basis for program planning.

Contents

The standard content headings are given below and the full details to be included under each heading are described in Appendix 1:

Background of the job

The job to be done

Data

Results

The job in operation

Organization for the computer

Processing

Take-on procedures

Future extensions and enhancements

Initial implementation estimates

Preparation and running times

Preparation and running costs

Appendices including file and print layouts, specimen forms, examples of printed results

Wherever appropriate, the information should be taken direct from the systems specification to save duplication of effort and to preserve accuracy.

Acceptance of documentation

The suite specification must be agreed with all departments concerned with the project in order to ensure that it is satisfactory from all points of view before more detailed work is started.

The people who must be consulted, and the reasons for their particular interest in the suite design are as follows:

The system specifier, in order to check that all requirements have been included and to ensure that links with other projects are possible.

The operating organizer, to ensure that the system is feasible from an operating point of view, that the required computer time is available and that the data preparation load is acceptable.

The user, to check that the time scales for preparation, take-on and normal running are acceptable. This will be a systems function regardless of who is responsible for the suite design.

The program planner, to ensure that the system is feasible from the programming point of view, for example, that it does not exceed the computer configuration capacity, and that the preparation and running times are feasible.

Once agreement has been reached, and the design accepted, no changes may be made without further consultation with those concerned, since what may appear to be an insignificant alteration from one person's point of view, may in fact cause major changes from another point of view.

Advantages of documentation

The main advantage of producing a written description of the system is that it can be formally checked and accepted by all departments concerned. Although such agreement can be obtained by discussion, without any formal document there is no guarantee that everyone has agreed to the same design since the presentation could vary from discussion to discussion.

The standard contents not only aid understanding but also act as a check list to the information which must be included, and therefore considered when designing the suite.

The concept of separating the systems specification from the suite specification means that different individuals can be assigned to the two tasks and use made of specialized expertise. Additionally the non-computer personnel, who are more interested in the requirements than how the computer system operates, will not be confused by, what is to them, irrelevant technical information.

Without a standard document which has been checked and updated by agreed changes, it is possible to reach the suite testing stage before it is realized that the computer system is inadequate from the user or operating point of view, or that there is insufficient computer time available to produce the results in the required time scale. At this stage there is not time to recover the situation and the result is continual pressure on the operating organization, a dissatisfied user, and a large amount of program maintenance in order to improve the system.

VETTING THE SYSTEMS DOCUMENTATION

There are two possible situations which cause the programming personnel to be involved in vetting parts of the system documentation.

The first of these occurs when the programming function includes the suite design and is therefore concerned with the vetting of the system specification.

The other occurs when the systems function includes suite design and the programming responsibility is to check that the design is practical. Whichever situation pertains, the vetting stage cannot be done in isolation and ideally the aims of the system, or the design, should be discussed between systems and programming during production of the documents so that the vetting stage becomes a formal acceptance of the final documentation.

Essentially this vetting stage marks the handover of responsibility for the computer system, from the systems analyst to the programmer. The return of responsibility is the point at which the systems analyst conducts the systems trials. The vetting permits the programmer to check the accuracy, consistency, feasibility and completeness of his brief.

This independent check that the overall systems design is completed gives an opportunity to refer back any queries to the systems analyst. As such queries may necessitate a further systems study it is imperative that they should be made before programming work commences.

Vetting the systems specification

Basically the final document should define all the requirements of the whole system. It should include:

Specification in complete detail of all the initial inputs and final outputs of the system.

Specification of all logical data which will constitute the files of the system.

Specification of all processing whereby input and data files interact to produce results.

Specification of any take-on procedures.

Where the logic of the system dictates it:

Specification of the order in which processing is to take place.

Allocation of data items to files.

Specification of sorts or other data handling methods at appropriate points.

These are the minimum requirements. Further details are given in Appendix 2. The suite designer should ensure that there is sufficient information to allow the further steps in the project design stage (suite specification, suite and file design) to be completed. Any mandatory constraints on the suite organization should be justified.

Vetting the outline suite specification

In those organizations where the systems specification is extended to encompass the suite specification, it will remain a programming function to vet the contents for consistency, completeness and feasibility. It is recommended that the programmer is involved in development of those sections which are program oriented. Any points of doubt should be resolved at this stage so that the system may be frozen before the program design stage begins.

An increasing number of systems analysts are adopting formal disciplines of specification (such as decision tables), and where such techniques are employed in an organization the programming department should be familiar with them. As the use of decision tables is so closely tied to programming techniques, this subject is discussed in some depth in Appendix 3.

Summary

As there will normally be an overall statement of requirements largely in non-computer terms, to permit study and acceptance of the scheme by the management, this will be a natural checkpoint even when the systems study and suite design are carried out by the same person.

Above all, as the programming department will bear the responsibility for the implementation of the suite, this will be their opportunity to accept the project or refer it back to the systems analyst.

SUITE DESIGN

In designing a suite the source document is the systems specification, and the result is the outline suite specification, incorporating a chart of the suite organization and initial programming estimates.

The overall task must be divided into programs which will function efficiently within the hardware and software environment provided. This means that the designer must have knowledge of these two factors. The operating considerations of the suite must be covered, an overall strategy for operator requirements and communications being defined. The data must be streamed into files to form the basic communications of the system.

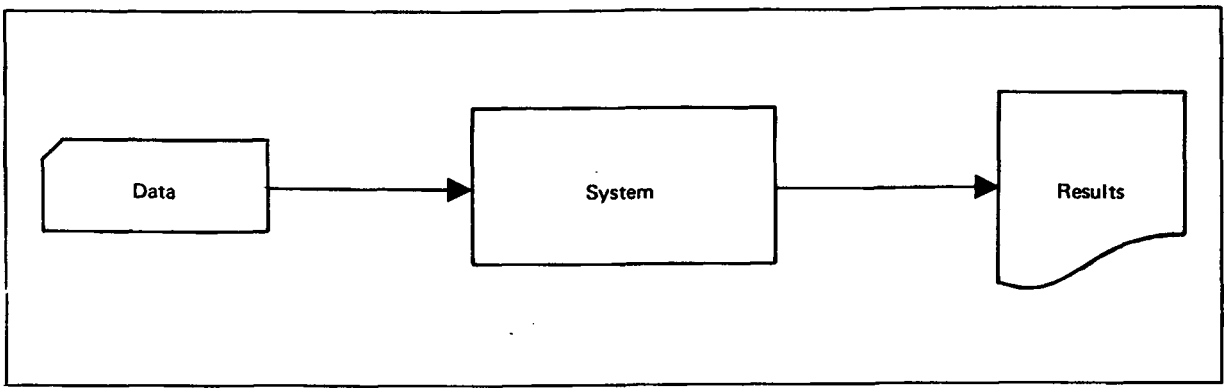
The design of files is a major part of suite design, but this section is restricted to a description of the allocation of files. A description of the detailed layout of files is given in the section *File design,* on page 15.

Principal aims

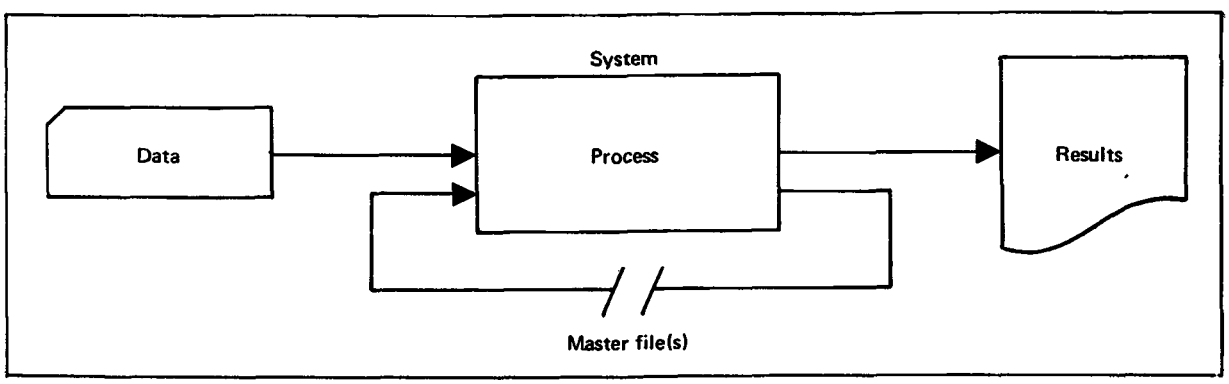
The first aim of suite design is to take the tasks to be performed and break them down into computer programs. The relationship and communications (files) between programs must be specified. When this division has been made, estimates are made of the cost and time required to produce the programs, and these matched against the resources available and the time requirements of the system. From this point the computer system is then modified with the aim of minimizing implementation cost and/or timescale, and maximizing the running efficiency of the system.

Overall concept

Essentially any system may be considered in the following way, with data in and results out.



This may be further expanded as follows:



Hence the first step in suite design must be to list:

All input, by type and times of availability

All main, or standing, file items

All tasks (interaction between input data, and main file data to produce results)

All results, by type and times required

The overall flow through the system may be charted in this way. The system can then be divided into programs, files can be designed and the system can be optimized.

Dividing the system into programs

Commercial data processing is principally concerned with the processing and maintenance of raw data and main files and the production of results files (eventually printed). The basic elements of any system therefore are almost certain to consist of the following standard units.

Type	*Name*
A	Data vet
B	File interrogation
C	File merge
D	File maintenance/update File set-up/takeon
E	File search/extract File edit File split

Type	*Name*
F	Media conversion, for example card to tape
G	Statistics
H	Sort

Once the necessary sequence of these units within the system has been determined it must be decided how they are to be grouped to form programs. For example, the first program of a suite will normally consist of:

A Data vet

\+ D File set-up (to set up a file of vetted data or it may be more complex and consist of:

A Data vet

\+ E File search to extract facts required by the vet

\+ D File set up

The designer must consider the advantages and disadvantages when deciding whether single-unit or multiple-unit programs are to be used. Some of the facts which must be considered are:

Single-unit programs

For

1 Small use of core store (which may enhance the multi-programming potential of the computer).

2 Small units may be written and tested more easily and efficiently.

Against

1 Intermediate files are generated to communicate between programs, aggrevating control problems.

2 The reading and writing of intermediate files will slow down the running of the system, and increase the load on the input/output hardware.

3 It may be operationally complex to run a number of small programs.

Multiple-unit programs

For

1 Passing data internally between units will save peripheral time thereby cutting the running time of the system and reducing the load on the input/output hardware.

2 A large program may save time in loading and setting up.

Against

1 As each unit will probably add an extra file to the program, this may strain or exceed peripheral resources or inhibit multi-programming.

2 A need for more core store may exceed the store available or inhibit multi-programming.

3 Large programs are complex to write and test (although modular programming techniques will overcome this problem).

Summary

The primary aim of the suite designer must be to meet all the requirements of the system, but he must do this within the limitations of the resources available, that is hardware, core and peripherals, and programming manpower. He must satisfy himself that the computer system will function efficiently. This may involve the specification of overall program design strategies: checks and reconciliations, use of standard programs, program overlay techniques. An overall data security strategy must be planned at the suite level. Further details of this strategy are given in Appendix 4.

This stage is one of the most vital in the implementation cycle. If work at this stage is not thorough no amount of hard work will build a good system within a poor framework.

FILE DESIGN

Files reflect the hardware and software resources and the data structure of the system. They therefore determine the programming implementation. Good file design is essential for processing information efficiently. If files are designed by the programmer as he writes the program, it will result in poor intercommunication between

between programs, and inefficient operation of the system. The main files and all those files which pass from program to program must be designed at the system level, as the specification of the individual programs will derive from the reading and interaction of these.

This section covers the principal decisions to be made and these are discussed in depth in Appendix 5.

Basic considerations

The principal points to be considered when designing a file are:

The selection of an appropriate medium

The position and relationship of each data item

Data formats

The ability to preserve the integrity of the data

The reading and maintenance of data files is the main process of commercial data processing, thus the efficiency of the file design will be reflected in the efficiency of the system.

Choice of medium

In almost all cases the media used for primary data input and for the output of results will be pre-determined by external considerations, and it will not be the problem of the suite designer to choose these. Hence the media to be decided upon are those used for files which are produced in one program for input to another, and main files. Where both are available this usually becomes a choice between magnetic tape an direct access media.

The mistake is sometimes made of thinking of direct access storage as superseding more conventional media. Every medium has uses to which it is particularly suited, and all peripherals available to the designer should be considered on their merits.

The designer should be familiar with the characteristics of the various devices and with the software which supports them.

Position and relationship of data items

The first stage is to determine in detail all the information which will be required on each particular file. The data is then grouped into related records which correspond to the function(s) of the program(s) which handles them. Within each record the format and layout of each data item must be determined. While there may be some overriding consideration, such as fixed length information preceding variable length, the relative positions of items should be related to any interactions between them. That is, the items should be grouped into sub-records which meet the requirements of sub-divisions of the processing task. Avoiding a dichotomy between the problem and data structure will considerably ease the structuring of a program, as will be seen in the section *Program design,* on page 23.

Once the record is designed the sequence of records must be determined. If a file is to be used by several programs this sequence should be carefully chosen to minimize the overall processing time. Where records are to be sequenced according to a key the key should be chosen to conform to the collating sequence of the computer in use to facilitate comparison of keys and sorting.

Any control records (heading, total, and end) should be specified.

Data formats

Data may be held in several ways within a record, for example binary, character. The best method for each item should be carefully evaluated, considering both field length and how the item is to be used. Data may be held in a record in three basic ways:

Fixed number of characters

Variable number of characters, with field separators

Integral number of words

Items held in variable-length fields occupy the minimum number of characters that their values demand, non-significant zeros and spaces being suppressed. Thus the read/write time is minimized but the processing time is increased by the need to expand or condense them in store. On a word-oriented machine this argument applies to all but integral word format.

A combination of formats may be the best compromise, with the most frequently used items being grouped in the most easily processed format, and less active items condensed. Obviously fixed length items will more conveniently precede variable items as then their position within the record will always be fixed.

The record is the logical unit of data, but it is physically read and written to and from magnetic media in blocks whose size may be dictated by hardware or software considerations.

Obviously there are three relationships between blocks and records:

Each block contains one record

Each record is spread over more than one block

Each block contains more than one record

The last of these is most common as the file processing speed is often proportional to the block size. Single record blocks are common where the program is dominated by the processor time or where the core store is limited.

The design of record layouts and the block size will automatically determine the physical length of the file. This is because there is always an overhead on each block written, which is the interblock gap, plus in the case of direct access devices, block identification fields.

The most significant effect of a large file size, whether due to a small block size or badly designed records, is that the file may be unnecessarily extended over a further tape reel or disc cartridge. This may be unacceptable, necessitating a redesign of the file, reducing the record length or increasing the block size.

Data integrity

Each file should be designed so that strict control of its use may be achieved. Data should only be accessible to those programs which are intended to process it, and where different versions of a file are produced care should be taken to see that only the correct version is accessed by each program.

All files must be capable of reconstruction in the event of loss or damage in any way. The system may specify that they are reconstructed from copies or previous versions or even that they are to be reconstructed manually, but whatever the method, the reconstruction method, its requirements, and consequences must be defined as an integral part of the system.

Advantages

The benefits to be derived from a correctly designed file are:

Running time is minimized

Core space is used to maximum effect

The file is acceptable to all programs requiring it

The communications of the system are presented in a standard manner for checking and programming

A well-defined file documents a system both from the definition of the data items and the relationship between them

Chapter 4 Implementation methods

This chapter deals primarily with two possible ways of satisfactorily implementing a programming project.

The two methods are deliberately presented as contrasts in order to highlight some of the factors which may influence the program planner, in the context of the next two chapters. In most satisfactory projects, however, a clear distinction would not be possible. Features of both methods are relevant to most circumstances. In Chapter 2 of Part 1 the impossibility of giving definite job description which would fit the circumstances of all users of this manual was pointed out. This allocation of responsibilities, taking into account the variable factors of the team, the project, the time scale, etc. is central to the method of implementation.

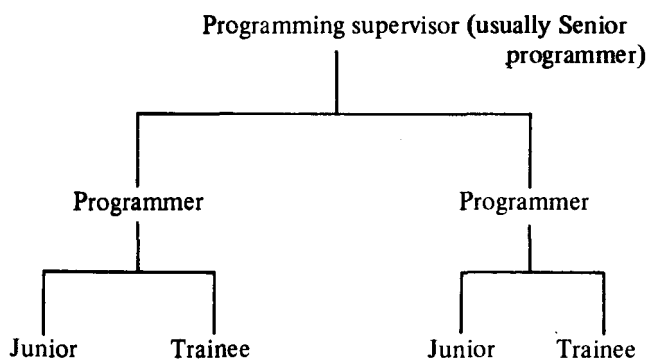
Decisions on the selection of the best method are the concern of data processing management and the material included is therefore only a summary of these two methods.

Recommendations on the basic method to be adopted are beyond the scope of this manual, and it is the ICL view that data processing management should be guided in this by company staff who can help to assess their specific situation. ICL's own programming services and specialist development groups ensure that possible methods are supported by reliable techniques, and are well understood before users are advised to adopt them.

Consideration is also given on the choice of language since this is to some extent dependent on the method of implementation.

THE CONVENTIONAL APPROACH

The conventional approach to implementing the programming tasks for a project is to set up a project team whose size will vary but is based on the structure shown in the diagram below.



Such a team will be responsible for producing all the programs for the project and in general each member, except the supervisor, will be responsible for one or more programs, and will write and test each of these as a logical entity.

Method of working

After the program specifications have been agreed, the team is set up and the programs are allocated to the team members. It is advantageous to divide the team into sub-teams, each responsible for an area of the suite so that certain supervisory tasks, such as checking, can be delegated and more than one person is familiar with each program.

Ideally each member of the team then produces and tests a complete program; however in a situation where trial time is scarce it is possible that a second program may be started before the first program has been completely tested; most experienced programmers can cope with this situation but this should not be expected of a trainee. Large programs can be broken down into several routines by the programmer responsible and more than one person can then be used to code the program; however the testing is carried out on the program as a whole,

although it is possible to design the test data so that testing can start on the main routines before all the subsidiary routines are written. This technique can help cut down the elapsed time for critical programs.

Advantages

The main advantages of this approach are:

1 Adequate supervision is provided and tasks can be delegated, without excessive specialization of functions.

2 More than one person knows each program thus giving cover for staff turnover, and providing for future maintenance.

3 Staff development is provided since each person carries out all the programming tasks for one program. Additionally the team concept allows juniors to become familiar with some of the project aspects.

4 It is reasonably flexible in that writing and testing of programs can be overlapped and programs can be divided among several staff when necessary.

Limitations

The main limitations of the conventional approach are:

1 In a situation of trial time shortage staff may be underemployed since it is not feasible to expect a person to deal with more than two programs at one time.

2 The programs may not be well designed since a proportion of the design may be a function of relatively inexperienced staff.

3 It is difficult to assess how much work has been completed. This is made easier by dividing the detailed programming into stages each of which has an end-point which may be checked.

4 If a program is delayed it is not easy to catch up on the lost time since all tasks are sequential.

Summary

The basis of the conventional approach is to consider, in general, that the program is the lowest level definition of a work unit. The programming tasks for a project are carried out by a team of programming staff and each team member is responsible for at least one program. This approach to programming implementation has been in use for some time and, although it does have some limitations, it can be used to implement work successfully and economically, with the side benefit of good staff development.

MODULAR PROGRAMMING

The basic concept of modular programming is that each program can be broken down into a number of small routines. Each of these routines is a relatively self-contained logical unit which can be written and tested as an independent entity. Each routine is therefore a more manageable unit and will be more easily understood by any level of staff. This method of implementation requires a permanent programming supervisor and one or more experienced people who are responsible for structuring the programs, defining the routines in detail, and finally linking the routines to form the program. The routines are written and tested by junior staff without design responsibilities who can be allocated on a more temporary basis.

Method of working

After the program specifications have been agreed the programs are allocated to the experienced programmers, who design the structure of the program (see Chapter 5, *Program structure,* on page 31) and produce specifications which define the requirements of each routine and their interfaces with other routines (see Appendix 7). These routine specifications are then allocated to junior programmers who are responsible for writing and testing the routine. When all the routines for a program have been tested independently, responsibility is returned to the experienced programmer who tests the program as a whole in order to ensure that the routines all link together as intended.

Independent testing

In order to test each routine as an independent entity a software aid is necessary. This software takes the form of a testing package which is used to read in the trial data physically, to set it in the interface areas of the routine and then to enter the routine, that is, the testing package simulates the entry conditions to the routine.

Advantages

The main advantages of this method of implementation are:

1 Junior programmers are given more manageable units to write and test, and therefore are likely to produce better results.

2 The implementation time of a suite can be shortened by allocating routines to as many staff as possible.

3 The logical design of the program is enforced and is carried out by staff with the necessary skills.

4 Programs are more flexible in that routines can be changed without affecting other parts of the program, and maintenance is made easier.

5 Routines can be written to perform general functions and a library of standard routines can be built up.

6 Routines can be written in the most suitable language.

7 If delays occur corrective action can be taken by using extra staff to produce some of the routines.

8 Progress on a program can be assessed more accurately.

Limitations

The main limitations of modular programming are:

1 More work is placed on the experienced staff in that they have to define each routine and the interface between them.

2 Only one person understands the complete program, since the junior programmers are concerned only with small parts of it.

3 Staff development for junior staff is difficult since they do not see all the program tasks.

4 Special training may be necessary in the use of the testing package and in the linking of routines.

5 If a testing package is not used, each routine must have its own coding introduced to set up the entry conditions. This coding would have to be removed before program testing.

6 There may be difficulties in linking routines together due to the structure of the software. This is only likely to affect second generation machines.

7 It is possible that more machine time will be used for testing since in most cases there is a basic overhead per compilation or trial whatever the size of routine.

8 As junior programmers become more experienced they will tend to be dissatisfied with writing small routines, and there may be a high staff turnover.

Summary

The basis of the modular approach is to consider that a small part of a program is the lowest logical unit.

The design tasks are carried out by experienced staff whilst junior staff write and test units whose relationship to each other is not obvious to them. This places great reliance on the thoroughness of the design and may inhibit the design awareness of junior staff. These factors must be balanced against the more obvious advantages.

CHOICE OF LANGUAGE

The language chosen for programming work can have an effect on implementation costs, the time taken for programming, and the operational running times. It is therefore important that the priorities of these factors are considered when selecting the language to use.

In a commercial environment the choice normally lies between a high level language, such as COBOL, and a low level assembler language. This choice will to some extent be influenced by the size of computer available, the power of the compilers, and the existing skills of the programming staff.

Whatever influences and priorities exist it is important that the programming language is given careful consideration, and as many factors as possible taken into account, so that best use if made of all available resources. Although in many cases the choice of language is outside the scope of a programming supervisor's responsibility, there is still a need for an awareness of the factors which influence the choice since ultimately the supervisor has to ensure that the language chosen is used to best advantage.

Areas of decision

There are basically four stages when a language may be chosen:

For an installation

For a project

For a program

For a routine

Obviously the more detailed the stage, the more flexibiiity there is in the choice of language, but unless suitable programming skills can be made available it is pointless to make a choice at a detailed stage.

High level languages

The selection of a high-level language, as opposed to a low-level language tends to produce the following situation:

1 Implementation times are reduced since the programmer has less detail to consider.

2 Use of computer time for trials is reduced since the program is less detailed and the compilers can carry out syntax checking; however, since compilation may take longer than assembly, the reduction in computer time is not in proportion to the reduction in the number of trials.

3 Maintenance of programs is easier since a higher degree of self documentation is inherent in the coding.

4 Staff are trained more quickly, since the use of language is less complex.

5 Object programs will use more core store.

6 Object programs may take longer to run since there are more instructions to process.

Low level languages

The selection of a low-level language, as opposed to a high-level language, tends to produce the following situation:

1 Implementation times are longer because of the amount of detail to be considered.

2 More computer time will be used for trials because of the level of detail.

3 Maintenance of programs is difficult unless sufficient annotation is included.

4 Training of staff takes longer due to the language complexity.

5 Object programs are more efficient in use of core store.

6 Object programs are more likely to have efficient running times.

7 Greater flexibility is possible in the use of the computer.

Mixed languages

The use of mixed languages allows the language appropriate to individual problems to be selected. Thus the advantages of the high-level languages can be achieved whilst the disadvantages can be minimized by use of low level languages for areas where machine efficiency is important. The disadvantage of this selection is that staff either need to be trained in more than one language, or they specialize with the result that resource allocation becomes less flexible.

The use of mixed language programming automatically implies a degree of modular programming as the routines in different languages must be compiled separately and interfaced.

Summary

The choice of language depends on what an installation is setting out to achieve, and the situation should be assessed carefully. For instance, it is no use producing projects in a short time if there is no computer time available to run them. Conversely a highly efficient project, in terms of use of computer time, is wasted if it is not operational on time. Subsidiary factors which influence the choice are those of staff experience, efficiency of compilers and the needs for compatibility between different machines.

The language should not be chosen, at any stage, on a basis of personal likes and dislikes but should be chosen on the basis of a careful assessment of the situation.

Chapter 5 Program design

This chapter covers the major factors which have to be considered when designing programs. The program specification, which is the end point documentation of program design, is described first in order to give a frame of reference for the other subjects. Each of the major factors is then described separately.

PROGRAM SPECIFICATIONS

A separate specification is produced for each program. The intention is to group together those details which pertain to a particular program, and to include additional detailed information so that the specification forms a complete description of the program structure and all the information required to produce the program, and provides a basis for maintenance.

Except in the case of very small projects (one or two programs) where it may be possible to produce programs direct from the outline suite specification, the program specifications must be considered as essential documentation for two reasons:

1 It is impossible for individual programmers to extract the necessary information from the suite specification without a danger of overlap or omission of some processes.

2 It is essential for maintenance purposes to know precisely where each part of the processing is performed.

Contents

The standard content headings are listed below and full details to be included under each heading are given in Appendix 6.

Suite introduction

Suite organization

Identification

Introduction

Characteristics

Tasks

Data

Results

Processing

Reconciliations and checks

Restart and re-run procedures

Error conditions

Outline flowchart

Trials plan

Appendices

Method of production

In order to produce the program specifications the program planner has first to study the outline specification and/or the system specification until a thorough understanding of the requirements and broad computer organization has been gained. Each program must then be planned and specified in detail taking into account restart and re-run procedures, checks and reconciliations, processing requirements, design of work files, and if necessary any multiprogramming considerations.

During this phase the program planner is likely to raise queries and may even require to make minor changes. Such queries and changes must be settled with the people responsible for earlier stages to ensure their feasibility and to update all documentation.

The program structure and overall logic must then be defined, and the broad trials requirements planned. If modular programming is being used for implementation it is also necessary to provide routine specifications (see Appendix 7) showing the interfaces between routines and indicating which areas of processing are applicable to each routine. The completed specifications must be submitted to the people responsible for the earlier stages so that the contents can be checked before any further programming work commences. Once the specification has been accepted a program file should be set up with the specification as its first content.

Advantages of documentation

The program specifications form the link between the program design stage and the detailed programming. The production of a formal document which can be checked ensures that the time consuming programming stage is not started with inadequate or inaccurate information and thus avoids wastage of effort.

The division of program production into two stages allows the more experienced staff to concentrate on the planning activities, and consequently programs are likely to be well designed and run efficiently, while the less experienced staff concentrate on providing solutions for the detailed problems.

The standard content headings are an aid in comprehension and therefore make the document more useful for maintenance purposes, as well as providing a check list of the information to be included. The standardized format also helps to impose a discipline on the program planner and enforces more orderly methods of production. This means that junior staff can be shown more easily how to plan and therefore aids practical training and staff development.

To expect junior programmers to write efficient programs without a program specification is not realistic and will result in clumsy, inefficient programs which are difficult to operate and impossible to maintain, and which may not even provide the required results. Whilst it is possible for an experienced programmer to do without a written description this should not be allowed since checking will not be possible, and there will not be any maintenance documentation.

RESTARTS AND RE-RUNS

It may sometimes be necessary to reprocess all or part of the data submitted to a program. This may be because of one of the following reasons:

1 A machine, program, or operator fault occurs during the run.

2 A further copy of the results or part of the results is required.

3 Part of the original results has been destroyed.

4 A system, program or operator fault has caused the run to be abandoned before the end is reached.

There are two cases to consider: firstly where original processing has terminated before completion, and secondly where the original run was completed but it is necessary to repeat part of it.

A restart enables processing which has terminated to recommence from a specified restart point and to continue to the end of the data.

A re-run enables processing which was completed to recommence from a restart point and to continue to a further specified restart point.

The general principle of restarting and re-running is to preserve sufficient information at certain points during the run, to allow all store and file locations used by the program to be set up at some later stage exactly as they were at these points.

If reprocessing is necessary it is clearly desirable that the time taken is minimized. This is achieved by dividing the data into groups, a group being the minimum amount of data that can be reprocessed at one time. The boundary between each group is a restart point.

Data groups

It is necessary to define the restart points in such a way that the processing time for the data occurring between successive points is of a satisfactorily short duration. It is suggested that something of the order of 15 minutes is

a desirable aim though on a small machine it may be uneconomical to provide restart points in a job which runs for less than 30 minutes.

In some jobs there will be a fairly obvious natural division of the data which can provide a ready made restart structure (for example Departments, Ledgers, Sales Divisions). In other cases it will be possible to make use of physical divisions in one of the data files (for example, a box of cards, a reel of magnetic tape). Further details on data groups are given in Appendix 8.

Restarts

A restart point (using either a software facility, if this exists or a user routine) should be inserted in any program that is expected to run for longer than 30 minutes.

At each restart point on the original run sufficient information must be output to one of the output files to enable the store to be subsequently set up in the same way, and to enable all files to be aligned at the same point as on the original run.

On restarting it is necessary to be able to input the file on to which the restart information was dumped and by using control data fed otherwise to align at the required restart point. After establishing the store as it was formerly and aligning all other files, the restart control file must return to use as an output file. It must however be optional for any output file to immediately force the end of a reel. This is mainly in case the original volume is unusable from some point onwards.

Details of restart techniques are given in Appendix 8.

Re-runs

The main problems with re-runs are implied by the need to terminate the processing at a point other than the end of the data. In general the output from a re-run must be to a different magnetic tape reel or area of disc from the original run. It may have to be because of physical destruction of the media. The initial setting up of the store and alignment of input files is much the same as for restarts but output files must be treated as separate logical files from the original files. If possible the marriage of the re-run output with the original run output should be accomplished by the normal reading of each file in the subsequent programs, for example by restart groups being separate reels of magnetic tape and the re-run output being a substitute for the original produced ones (which may no longer exist). If this is done there must be a distinction which is clear to the operators, between the identities of the original and re-run reel. However, the distinguishing feature must not invalidate the acceptance of the re-run reel as part of the original file when the file is subsequently read.

Alternatively one may have a special merging program which reconstitutes a complete file from the original and the re-run. In the case of serial files this will only be acceptable for relatively short files where the running time of the reconstitution program will be small. This technique also overcome the problem of accumulating overall totals.

A further requirement of re-runs is to ensure that, collectively, the original and the re-run have processed all the data. This may seem straightforward but it implies a close inter-relation between the reconciliation system of the original run and that of the re-run.

Ideally it will be possible to restart during a re-run and still terminate at the right place. Also it will be possible to re-run several groups at a time, not necessarily consecutive groups. Details of re-run techniques are given in Appendix 8.

Summary

Not only do these techniques minimize the duplication of computer effort (at a real cost of £'s per hour) in the case of failure of some sort, but the elapsed time to complete the interrupted processing is minimized. Whatever the machine size this saving may be the only economic way of coping with faults, the cost of providing the necessary routines being weighed against the savings or the continuing security.

DESIGNING CHECKS AND RECONCILIATIONS

It is the responsibility of both systems designer and the program designer to ensure that checks and reconciliations are built into all programs to minimize the processing of incorrect data and the incorrect processing of correct data. When the systems designer has specified these checks, it is then important that the program designer should satisfy himself that the checks specified are adequate and useful. All checks should be included in program specifications and not left to the discretion of the programmer.

Incorrect results may be produced for any of the following reasons:

Hardware error

Software error

Operator error

Data error

User program error

Three types of check are used to supplement those which are carried out by the hardware and software.

Reconciliations : these are used to detect hardware, software and user program errors.

Data vetting : this is used to detect errors in the data, whether arising at source, data preparation or computer reading stage. Data is checked not only for feasibility, but also to detect missing or duplicated data. Incorrect prime data should never be permitted to cause an unscheduled halt to a program.

Checks on operator action : some of these may be done by the software, but others will be included in the user program.

Each type of check is considered separately.

Reconciliations

The basic purpose of reconciliations is to ensure that data which has been accepted by the computer system is not lost, duplicated or corrupted. This could be done most easily where the final output should correspond completely to the initial input; unfortunately, most computer projects are more sophisticated than this and reconciliations have to be designed individually for each file or program.

Any programmer who feels that with present machines and operating systems, reconciliations are superfluous should reflect that by far the most frequent cause of reconciliation failure is program error. Nevertheless the course adopted for reconciliations must be a compromise, based on weighing costs against usefulness. Reconciliations should be designed to detect an error at the earliest point possible, with the aim of minimizing reprocessing.

In general, there are three types of reconciliation:

1 Validation of the contents of a file held on backing store

2 Validation of an updating process

3 Validation of a calculation process

Details of these reconciliations are given in Appendix 5.

Data vetting

No assumptions may be made about the correctness of data which is prepared outside the computer or is carried on an unsafe medium (paper tape or cards). It is customary for as many feasibility checks as possible to be gathered into a data vetting program to avoid the need for later programs to check the data before processing it.

Feasibility checks include checking that each character is a field is valid for the character range and radix of that field, that the correct number of fields are present, that each field lies within the range of possible values, that check digits are valid, and that the contents of different fields (in the same or different records) are consistent with one another. Checks of this type should be made as comprehensive as is consistent with efficient use of a computer and failures should be reported intelligibly.

To guard against feasible, but incorrect data, input control totals should be prepared manually for important quantities and values; these control totals are then checked in the data vetting program. To minimize the possibility of errors in the manually prepared total, data should be batched into fairly small units and separate control totals prepared for each batch before punching. Batching also makes it easier to locate reported forms and to detect forms which have been omitted or duplicated at the data preparation stage.

To provide further safeguards against missing or duplicated data, information should be included in the data to inform the data vet program how much data it should receive. Multi-level batching systems will simplify this problem.

For example, for cards:

Divide data into sequentially numbered boxes.

Divide boxes into batches (sequentially numbered if this can be easily achieved).

At the start of the first box place a card stating the number of boxes submitted.

At the start of every box place a card stating the number of batches contained in that box.

At the start of every batch place a card stating the number of cards contained in the batch and giving control totals for important values and quantities.

Checks on operator action

The best way to eliminate operator errors is to design a system which is simple and straightforward to operate and for which clear concise operating instructions are provided. Operator decisions must be minimized and those which are unavoidable must be capable of being made by off-line operating before or after the run, not by on-line operating during the run. As far as possible, all operator actions must be checked; the senior programmer must ensure that this is being done, either by the software or by the user program. A very common example of this procedure is the check performed on a file, after loading, to ensure that it is the correct file.

To simplify the operator's job, the amount of control data to be submitted by the operator at run time must be kept to a minimum. The computer system must be designed in such a way that as many job control parameters as possible are held as part of the program library.

Summary

Not only must the programming power of the machine be used to guard against the diminishing threat of machine error but checks must be included to detect any of the following errors caused by human failure:

Software error

Operator error

It should always be assumed that operators and users will make every type of error. Provision must be made to detect and correct the assumed errors. No data error should ever by responsible for any unscheduled interruption to a program. Routines should be provided in data vets to recognize every possible error.

Not only will these measures force complete testing of any system, but the knowledge of their existence will improve confidence in the operational suite.

MULTIPROGRAMMING CONSIDERATIONS

With any capital equipment the aim must be to make maximum use of one's investment, and with a computer this means making maximum use of central processor, core store and available peripherals.

On a powerful computer few programs will be expected to make use of more than a fraction of the central processor time available over the extend of its running time, or to employ all the peripherals available. Hence to increase utilization of the computer one must multiprogram, that is, concurrently run a number of programs with complementary requirements in terms of hardware resources.

For example consider the following programs:

A — A print program, with small processor requirements, but relatively long running time, because of its peripheral usage.

B — An update program which might be considered as average in all respects.

C — A statistical analysis program, with small peripheral requirements, and large processor requirements.

Considering these programs separately the running time might be:

A 35 mins.

B 20 mins.

C 15 mins.

TOTAL RUNNING TIME 60 mins.

Running concurrently the processor requirements would interleave and the times might be:

A	40 mins.
B	35 mins.
C	30 mins.

TOTAL RUNNING TIME 40 mins.

The individual running times are increased to some extent, but as they all start at the same time, they are all complete after 40 minutes. Thus it is apparent that a multiprogramming capability increases the potential of a computer considerably.

If programs are to be run concurrently, it is necessary to schedule jobs in order to make good use of multiprogramming. This is a complex problem and may be simplified if the scheduling factors are considered at the systems and program design stages.

Scheduling

When a computer is first installed, the work load is small and scheduling of work might be simple and of minor importance. Later when the load increases scheduling may become vitally important, and some planning for this situation at the start will simplify matters enormously.

Software can schedule programs by matching program requirements against available resources, but a scheduler normally examines the job queue program by program. The first program which may be loaded is not necessarily the program which should be loaded first. When deciding which program to load next the overall situation must be considered. In particular the probable running time must be considered, as well as the matching of requirements against resources.

The difference between an optimum solution and a good manual solution to this problem may not justify the cost of full computer scheduling and rescheduling with regard to constantly changing conditions. The scheduling facilities offered by those operating systems which handle this task provide an effective solution. The operations controller, however, can adapt rapidly to changes in conditions and requirements, and can increase efficiency, provided that the number of variable factors is limited.

The variable factors may be limited by using the following scheduling methods. These methods may be used singly or jointly.

STREAMING

Divide the machine into streams and schedule each stream separately. At any one time only the limited resources of one stream and the programs for one stream need to be considered.

PROGRAM KITS

Decide on a limited number of program configurations and design all programs within these restrictions; for example, class A programs use two magnetic tapes and not more than a 10K store.

BATCHING

Schedule work as a number of independent batches. This means that only a limited number of programs need to be considered at one time and since the running time of a batch is short it is easier to schedule.

All these methods reduce the potential efficiency of the machine, but used intelligently will increase the actual efficiency by bringing the problem within the scope of the scheduler.

The problem may also be simplified by the use of visual aids such as bar charts, block charts, pegboards.

Stream profiles (System 4)

Common sense must be used in the application of the various techniques. Streaming is only useful when all streams remain active. If a stream is not being used its store space could probably be used profitably to increase the multiprogramming potential of another stream. Thus if the load is variable the stream profile may be varied from session to session. However, the design of programs is simplified if the streams used consist of a limited standard set.

Example

	Session 1	*Session 2*	*Session 3*	*Session 4*
Store	Input stream	Print stream	Main stream	Input stream
	Print stream	Sort stream	B	Print stream
	Main stream		Main stream	Main stream
	A		A	B

When initially designing streams the following formula may be used.

$$\Sigma \text{ (stream size -2K bytes)} = \text{store size - Supervisor size}$$

In this way when programs are written and found to exceed the permitted stream size, there is a margin for expanding the stream sizes.

Program mixes

Some mention has already been made of the need to choose mixes of programs with complementary processor requirements for multiprogramming. Not only must the input/output requirements be complementary but also the peripherals and the device control unit and channel requirements.

It is foolish to attempt to multiprogram a number of jobs which are principally reliant on a single heavily-loaded piece of equipment, such as a single tape or disc channel, for example.

Care must be taken in multiprogramming programs using replaceable discs to ensure that the total number of different disc packs required at any time does not exceed the total number of the disc transports available.

Care should also be taken to see that the sharing of a disc cartridge between programs does not result in unacceptable delays due to extra head movement times.

Off-lining

Two basic rules are often employed in planning multiprogramming systems:

1 Make maximum use of the most heavily loaded equipment.

2 Run the largest programs in the shortest possible time.

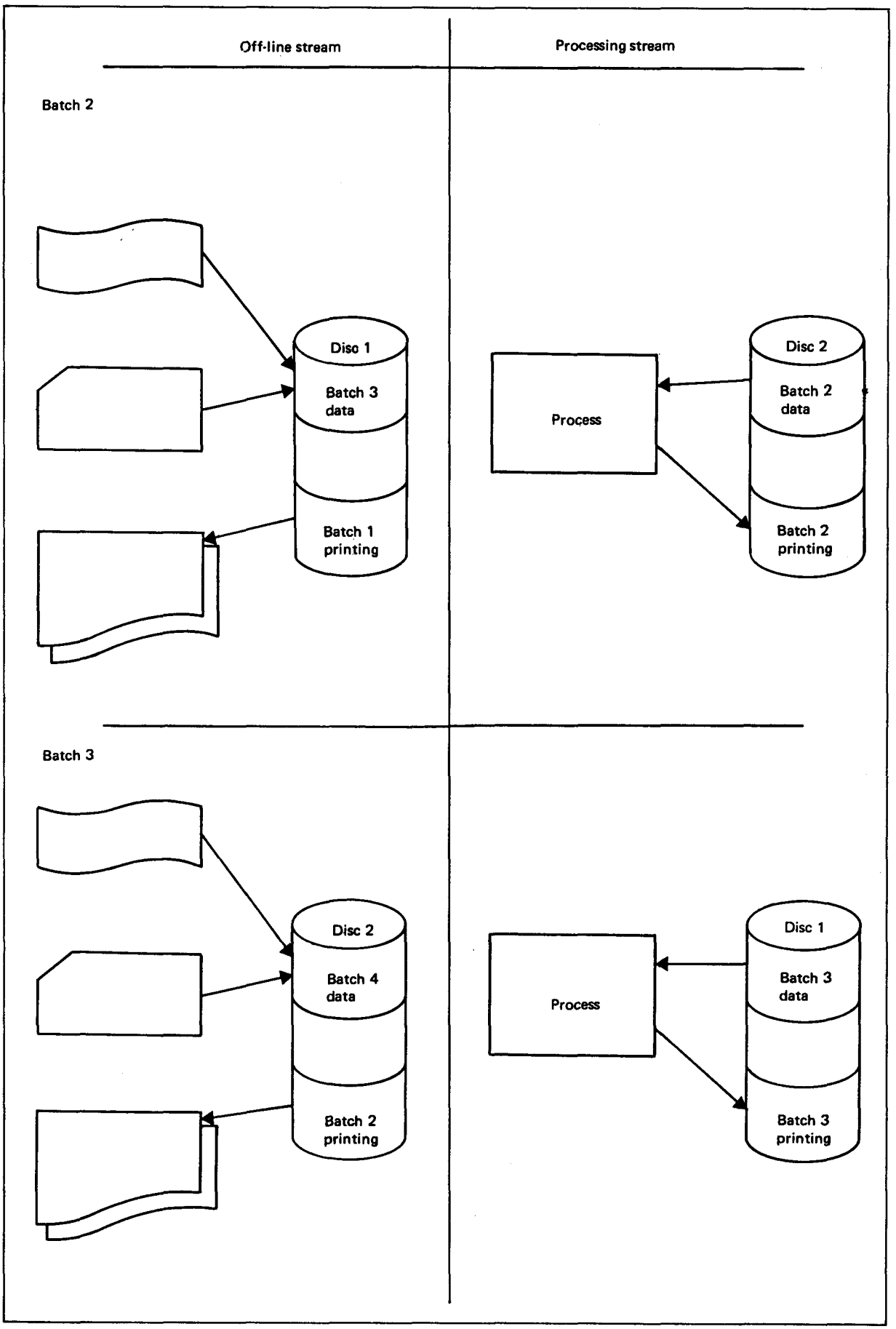
As the most heavily loaded equipment is often a single overworked line printer the situation may be alleviated by not permitting programs which cannot drive the printer at maximum speed (programs with low volume printing) to print on-line. Further, rule 2 may be achieved by removing slow device dependence from main (for example update) programs.

Both these aims lead to the idea of off-lining all primary input and printing, that is, copying all cards or paper tape to disc or magnetic tape before processing and off-lining print files to tape or disc for subsequent printing. Obviously this technique may only be adopted where there are sufficient tape or disc transports.

A system of batching this off-lining using a single disc drive for primary input and printer output is illustrated. The head movements between the various files are tolerable as it takes some time to buildup for output a block of data.

Summary

With a powerful central processor the throughput may be enhanced by making full use of the multiprogramming potential. Proper use, however, is often only made if this technique has been considered at the design stage, and the programs designed to be complementary.


Off-line stream
Processing stream
Batch 2
Disc 1
Batch 3
data
Batch 1
printing
Process
Disc 2
Batch 2
data
Batch 2
printing
Batch 3
Disc 2
Batch 4
data
Batch 2
printing
Process
Disc 1
Batch 3
data
Batch 3
printing

Even on a computer too small to be able to utilize full multiprogramming the machine load may be reduced by the use of off-lining of raw data input and printed results.

PROGRAM STRUCTURE

In planning the structure of a program the aims must be to plan the program so that:

It may be implemented efficiently

It will be easily understood

It will run efficiently

In the section on modular programming, the advantages of a modular approach to programming implementation have been listed. Here is described how a modular program structure is automatically and necessarily derived at this stage, even if modular implementation is not used.

Outline flowchart

The structure of a program is normally determined at the program specification stage, when an outline flowchart is produced. The outline flowchart represents the sequence of tasks of the program at an elementary level:

The reading of files

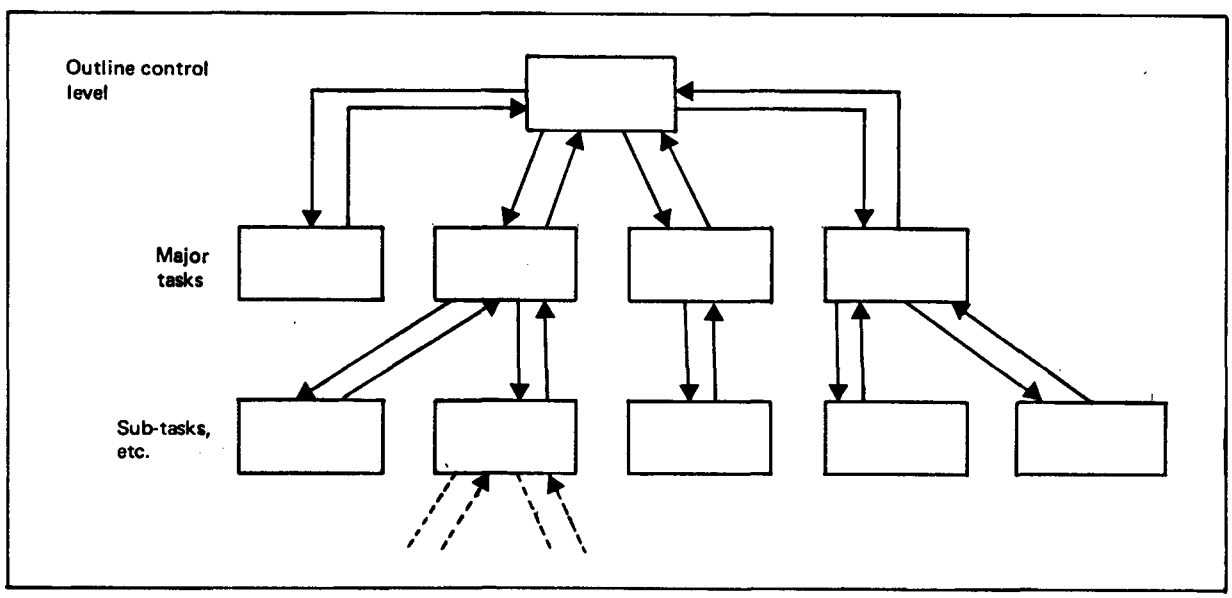
The correspondence of files

The processing of tasks

The writing of files

When the program has been charted at this level the program will have a start and an end, and consist essentially of file handling. By adopting the flowcharting standard, (as described in detail in Appendix 12), each box of this highest level flowchart is expanded as necessary. At the lower level individual flowchart boxes are further expanded, and the process continues until the logic of the program is completely defined.

In representing the program at any level as a number of discrete boxes the program is divided into a number of processing areas or tasks, that is, modules.



Thus the flowcharts automatically impart a hierarchical or levelled modular structure to the program. In order to achieve a modular implementation potential the division between individual boxes must be at a logical point.

Problem structure

In dividing the system into programs the system is divided into tasks whose sizes largely match the hardware resources. In structuring a program it must be further sub-divided into tasks whose sizes match the programmers' ability. For future maintenance the maximum size of the task should be related to the average programmers' ability.

In dividing the system into programs the files may be defined as the flow of data between them, and to cut running time the size of these files is minimized. In dividing the program an attempt should be made to minimize the flow of data and references between sections, to facilitate implementation.

As the program is a logical subset of the system, each module must be a logical subset of the program. To summarize the criteria for program design, a module should:

Be of a size related to the comprehension of the programmer producing it

Perform a complete logical function of the program

Have a minimum of reference to other routines

When dividing a program into logical units, that is, flowcharting, it falls into a series of levels. At the highest level, a commercial program is concerned with the reading and writing of files, and maintaining program totals. At a lower level it is concerned with the processing of each file and at successively lower levels with the processing of individual records within those files, individual sub-records within those records, and individual fields within those sub-records. Hence at each level there occur more and more logical entities than at the higher levels. A structure such as that described is a data oriented program structure.

On the other hand it is possible to consider program structure in an entirely different way. The overall problem can be considered as a series of sub-problems. For example, a pay calculation program may have the following tasks:

A	Carry out file control checks
B	Calculate pay for individual employees
C	Update running totals on employee records
D	Produce payslips

Then if any of these is complex enough to be larger than the maximum unit of comprehension it in turn may be divided into further logical entities.

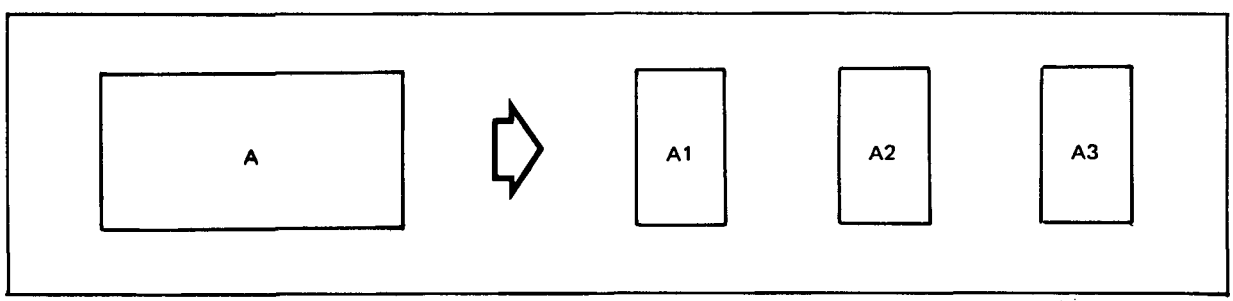
For example, the B may be further divided into

BA	PAYE
BB	Standard deductions
BC	Calculate free pay etc.

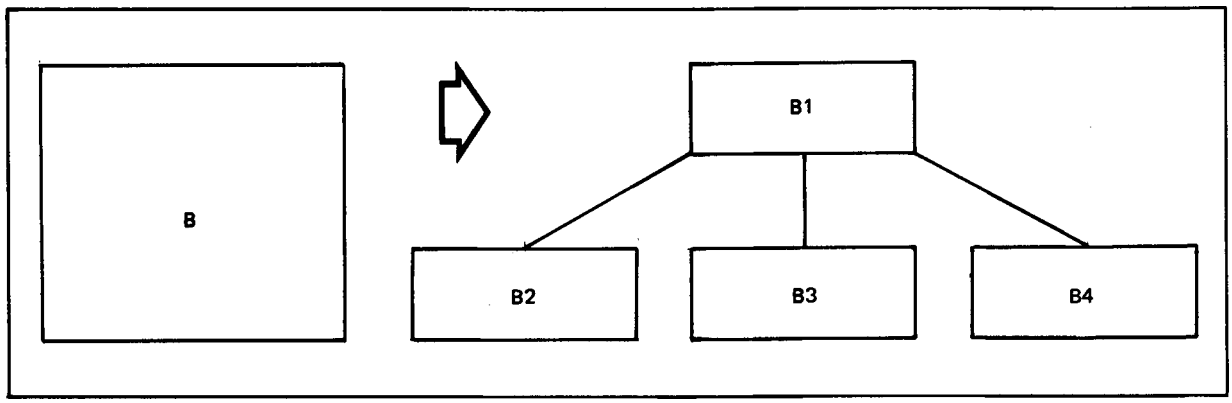
A structure based on such a principle is a problem oriented program structure. Either approach may be made to the problem of structuring a program, but if the files have been designed properly there will normally be a close correspondence between problem structure and data structure and both methods may be used together.

It may be seen, then, that at each level a program will fall into a certain number of parts. If any of these parts is too large it may in turn be divided into further parts at a lower level until the individual units satisfy the first criterion for program design. The third factor, intermodule references, should automatically be considered when deciding what constitutes a logical entity.

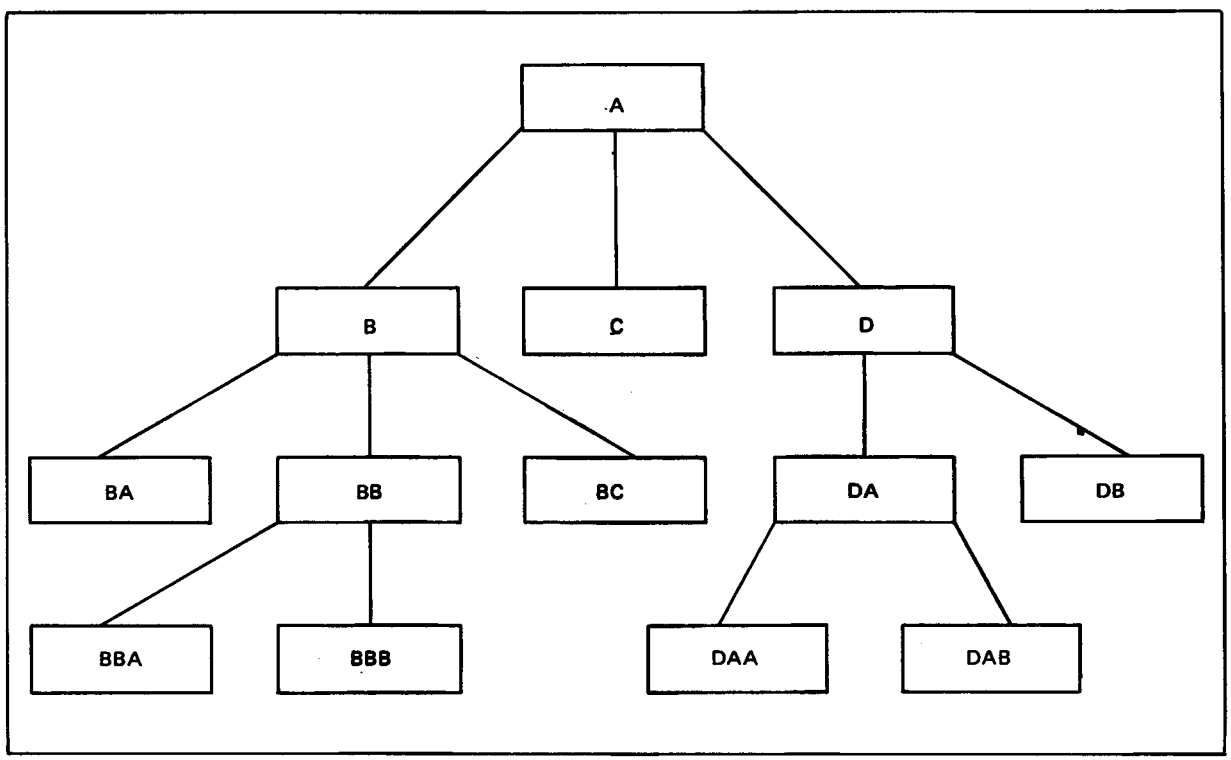
A logical entity on one level may be considered as several logical entities at a lower level. It would be inconsistent if a module could be divided into a number of independent parts, as shown in the illustration below.



However a module may be divided into two or more levels, the module at the highest level retaining referencing and co-ordinating functions:



Thus, while the modules at any single level are mutually independent there is a dependence across levels. However, by dividing the program in this way each routine is dependent on only one routine at a higher level. If the testing of modules starts at the lowest level in the program and moves upwards references are limited to a single link at each stage.

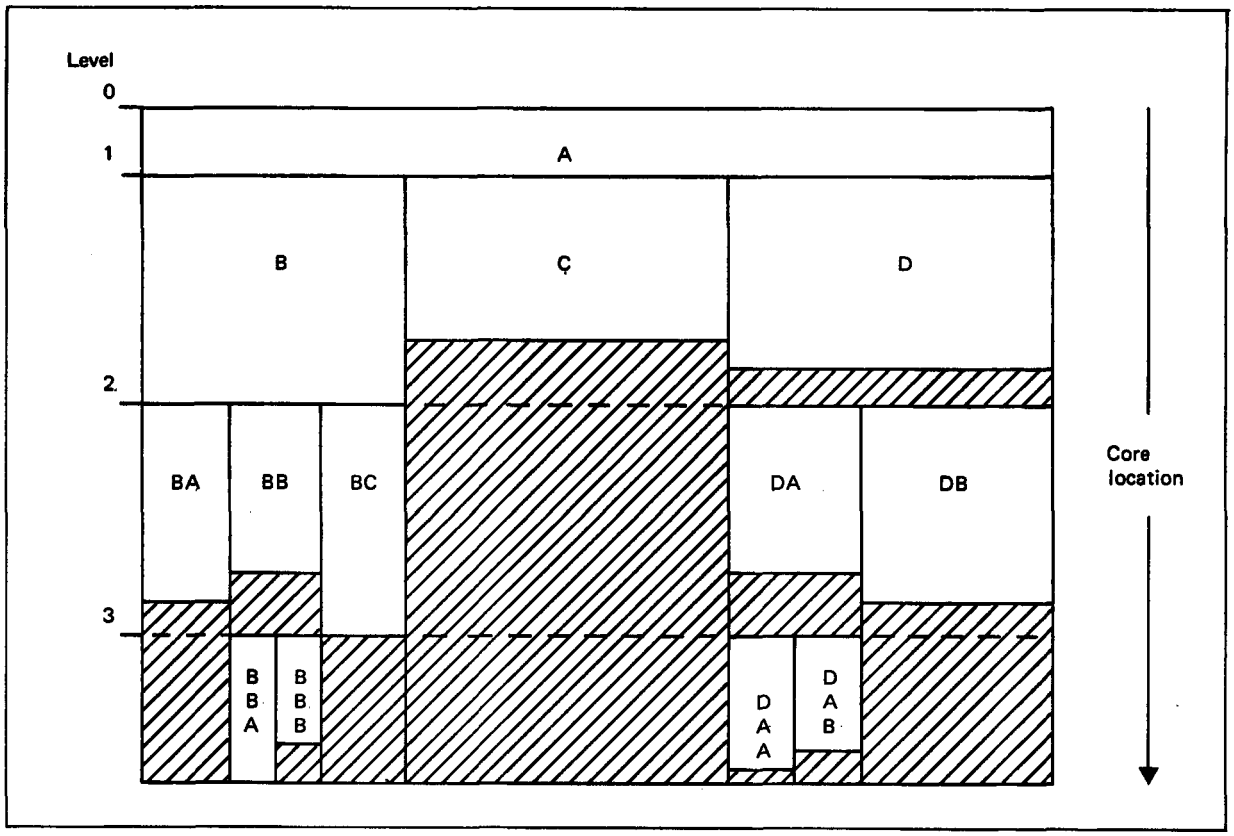


This levelled structure of a program corresponds directly to the standard method of flowcharting, where each box represents a logical unit which may be further expanded at a lower level.

Overlay techniques

When the program is divided into modules use of overlay techniques is facilitated. Strictly speaking, as there is no communication between modules at a single level all the modules at one level can be overlaid. The previous example might give a store layout as illustrated overleaf.

Under most circumstances however this would be uneconomical due to the time taken to call modules into core each time one was to be entered. The circumstances under which overlay techniques are employed will obviously vary, from the case of holding the modules on direct access device where the overlay time will be measured in milliseconds, to the holding of overlays on magnetic tape, where the search time could run to minutes.



In the former case overlay techniques may be used to effect useful economy by overlaying less active routines. On a data vet, for example, 40% of the program might easily deal with 90% of the records input. The overlay time to call in routines to deal with the other 10% might well be acceptable.

Slower overlay methods would probably be restricted to those cases where it was essential in order to accommodate the program with a limited core size. In this case the points of overlay would be determined such that each overlay was only called once or twice in the run of a program, if possible.

However the technique is used, the independence of the modules is the same independence required of program overlays, so the approach to program structure detailed above will serve to facilitate the use of this technique.

Summary

The obvious advantage of a modular approach to program design and structure is that attention can be focussed on the comprehension and implementation of a logical unit of coding, related to the programmer's capabilities rather than the computer's. A complete set of processing notes may be written for a module rather than an incomplete set for the program as a whole. The greater degree of self-checking in this scheme is obvious.

Further, the problem, and hence the program, is made easier by relating the data structure to the problem and the program structure to the data.

Even if a conventional, rather than modular, approach to programming is employed the advantages listed still accrue from structuring the program in this way. A logical and easily understood program layout should be achieved.

TRIALS DESIGN

It is necessary to design the trials process at an early stage in a project. This is to enable the best use to be made of resources and to give identifiable aims to the stages of testing each program. There will thus be minimum delay and uncertainty in implementing trials. Structuring the trials to the functions of the program to be tested follows logically from the planning of the program structure.

Trials strategy

There are two choices when deciding a trials strategy. The two choices are the conventional program testing method or the modular testing approach. The latter consideration is of course dependent on whether a modular programming concept was used in writing the programs.

Even using conventional program testing methods, partial modular programming is still available. Consider a data vet which has different processing routines for different types of record. Trial data can be selected which will just test the appropriate processing routine. This means that each processing routine can be tested semi-independently. Each section of the program which is to be tested as an entity is called a test unit.

If large programs or units of programs are to be tested, then it is advisable to plan various trial stages. Each trial stage should be defined to achieve a specific objective.

Example

Compilation (Trials stage 1)	The format of the coding only is tested at this stage. The program should be fairly clean before dry-running commences. This stage includes the formation of test units.
Basic (Trials stage 2)	The purpose of this stage is to test comprehensively all external communications (operator, file handling, etc.) as this will make all further testing easier and more straightforward. Also anything that will aid later processing should be tested (self-checking routines, reconciliations etc.). This stage can be broken down into a number of short independent runs, some of which may require a certain amount of temporary alterations to the program.
Extensive (Trials stage 3)	The majority of the paths through the test unit are tested using correct trial data.
Error (Trials stage 4)	The limits of items are tested and incorrect trial data is used to test all the error paths.
Complex (Trials stage 5)	The purpose of this stage is to test combinations of data and circumstances and generally to fully test the program using a volume of mixed data.
Cycle (Trials stage 6)	The main file is cycled if applicable.

In additional or alternative trials stages are to be used on a project then they must be defined for that project. This is a form of modular testing, the objective being to reduce tasks to smaller more manageable units which can thus be performed more efficiently.

There are three methods of using trials stages:

Individual testing	Each stage is tested separately. The program is then tested to ensure that the program will work for the different stages collectively as amendments for one stage may have corrupted previously proven stages.
Incrementive testing	The first stage is tested and when it is working correctly, all data, results and documentation are kept for future analysis. The data for the next stage is added to the existing data and on subsequent runs, the results from the previous stages can be compared with the correct results either manually or by using a standard program.
Decrementive testing	In testing the first stage, the complete set of trial data is used. Thus further testing may be gained without detracting from the purpose of testing the present stage. When a stage has been completely tested, the trial data for that stage should be removed from the trial data pack. The program should be finally tested with all the trial data to ensure that the stages are collectively correct. Alternatively, the stages may be retained and re-tested at every trial.

There are two overall methods of tackling program testing. Firstly trial data is designed to test individual programs. When the programs in the suite are all tested, trial data is designed to test the linking of the programs in the suite. The second method is to combine those two stages. Trial data is only designed for the first program in the suite (for example, data vet) and the master files. The data is carried forward to test all the programs by the intended process of the suite. This latter method demands tremendous foresight in the original trial data design and also obviously prolongs the testing of the suite as testing is in series. In the first method, testing is mainly in parallel and discrete entities of trial data are easily designed, so is thus the more common method of tackling trials. The second method should only be considered for a small and uncomplicated suite.

Trial data

In order to ensure that the programmer devises a comprehensive set of trial data to test his program fully, the basic guidelines for the trial data should be designed at this stage. Basic combinations of circumstances and sequence of events to be tested should be defined. From this initial design, the programmer should be able to develop the full detailed set of trial data. However, additional guidance on this task may have to be given to the less experienced programmer. By thorough planning and guidance in this way, a program can be proved to everyone's satisfaction. This is obviously not affected by the project or size of installation.

The final detailed set of trial data must aim to test the program fully. Within the limits of practicability, every single instruction and path in the program should be tested, although in fact it is the variations of conditions and combinations of paths that will fully test the program and prove that it meets its requirements. It is these factors which cause the most difficulty. With modular programming the problem is alleviated by the fact that a module is a fairly small compact piece of coding. Thus it is easier to conceive the trial data which will meet these needs. In testing large programs, this simplicity can be simulated by breaking the task of testing down into a number of stages, as outlined in the section *Trials strategy* page 35.

It may be convenient, when designing trial data, to use a testing schedule form (see Appendix 11) to note each significant circumstance and combination to be tested. Judgement is involved here as it is not usually feasible to note every single circumstance and it is virtually impossible and probably unnecessary to test every possible combination out of the full set of all possible circumstances. This trial data should not in general be used for linked trials, but it is useful to use the same master files.

File housekeeping and security

The design of the file housekeeping and security system for trials will depend to a certain extent on the installation and the type and size of the project.

File housekeeping means the maintenance of files in an orderly state, that is, on a given medium, unwanted or corrupt files or programs are deleted. By regularly maintaining the medium, optimum use can be made of it, since unnecessary references to files and programs are eliminated. Because of the continuous updating process during trials this will be fairly frequent.

Security means the keeping of duplicate files and programs. The use of these will minimize the time taken to recover to the current position in case a magnetic tape, for example, is corrupted or has parity errors or scratches, or is lost. The machine time used is also minimized.

Both the file housekeeping and security factors have to be offset against the programming effort, the machine time and media usage in setting up duplicates and submitting file housekeeping runs. A compromise is necessary here in evaluating the amount of maintenance and security needed against the effort and cost involved.

Summary

By designing trials in advance, they can commence smoothly and efficiently. The testing position should be recognizable at any time.

By designing the combinations of circumstance to be tested, it will ensure that the program will be fully tested. Thus the onus will not be totally on the programmer, although he will have to design additional trial data.

In designing the security of the trials system, the number of tapes and discs needed will have been calculated. This will enable them to be ordered and available in good time for trials.

Thus the planning and designing of trials is time well spent.

Chapter 6 Resource planning and control

ESTIMATING

Estimating is the first stage in the planning and control of resources. Without an estimate of the implementation time required it is impossible to establish a plan, without which there can be no control.

It is necessary for estimates to have a basis which will allow for improvements in future estimating and an objective assessment in the event of deviations of actual progress from estimated progress, for instance, it is essential to know whether an estimate is inaccurate or whether staff are inefficient.

The ideal basis for estimates must therefore be a unit of work which can be assessed originally and measured more and more accurately as work progresses.

Although estimating is an essential task, irrespective of size of installation or project, it cannot be regarded as an exact science and the accuracy of the assessment will be influenced by the experience of disciplined estimating, the ability to refer to records of previous projects and the judgment of the estimator. The programming supervisor must therefore ensure that the estimates are checked and also ensure that all factors taken into account are recorded for use in any future analysis.

Method of estimating

A recommended method of estimating, which will fulfil the requirements stated above, is to build up the estimate in stages as detailed below:

1 Assess the work content of the task, in standard units, and independent of staffing. The work content is based on the number of instructions required to program the task, and the complexity of the logic and functions to be performed by those instructions, in relation to a nominal average complexity.

2 Using the work content as a basis, estimate the man-days effort required to complete the task, assuming a programmer of average ability. The average ability must be defined according to installation recruitment and training standards.

3 Assess the ability of the programmer assigned to the task and adjust the man-effort accordingly.

4 Estimate the time scale required, taking into account overhead activities, the programming environment and the computer time availability.

5 Using the work content as a basis, estimate the number of trials and computer time required.

The assessment of the work content is of prime importance, since it forms the basis for the other stages, and the estimator must ensure that all the information available is taken into account and that the task is broken down into sufficient detail to make the estimates as accurate as possible.

Estimates have to be made at different project stages and the amount of information available will vary. For example it may be necessary to make an estimate during the suite design stage in order to assess the probable work load and end date. Such an estimate will not be very reliable and it is important that all assumptions made are recorded, and that the estimates are revised as more information becomes available. Further details of the estimating method are given in Appendix 9.

Summary

Estimates give the basis of progress control and it is therefore important that they are made as accurately as the information available allows. All factors taken into account and any assumptions made must be recorded with the estimates so that the effect of changes can be assessed. Estimates must be based on a measurable unit so that an objective assessment of variations from plan can be made.

A formal estimating method helps in training staff to improve the accuracy of their estimates and thus improves the reliability of the target dates. Without such a method estimates are likely to be subjective and vary widely, and planning and progress control will be ineffective and unreliable.

SCHEDULING

Scheduling is the establishment of a plan against which progress can be measured. The plan is based on the estimates of elapsed time and the objective is to schedule in such a way as to make best use of resources in achieving an acceptable end date.

Establishing a plan is an essential task, on any project, for several reasons:

1 The plan forms the basis of progress control.

2 If an end date is already fixed it is necessary to establish the resources required.

3 If the resources are limited it is necessary to calculate the end date.

4 If the end date and resources available are both fixed it is necessary to check the possibility of achievement.

5 It is necessary to know when computer time will be needed and what turn-round is required.

6 A balance between time, resources and cost can be achieved.

Any plan must of course be subject to revision if circumstances change in order to establish new end dates and make control realistic.

Method of scheduling

Schedules are best presented in bar chart form and in order to achieve a good balance of time and resources it is first necessary to draw rough bar charts showing the loading on programmers against time.

Known holidays and training should be entered, and the programs loaded bearing in mind the experience of the staff and the need to achieve a balance work load. Once a possible loading has been made the plan should be checked for restrictions of end date, resources, cost or computer loading. It is essential that any failure to meet these restrictions is discussed with management, so that alternative plans can be made.

The schedule should then be redrawn showing programs against time, for use in progress control, (see Appendix 10). This schedule should include suite testing.

Scheduling considerations

These are no firm rules which can be followed in establishing a plan but factors which should be considered can be given for guidance:

1 Divide the team into groups, each dealing with a part of the suite. This will help in delegation of checking and will ensure more than one person knows the programs and will thus make it easier to reduce the team towards the end of the project.

2 Suite testing can be overlapped with final program testing provided that data vet programs are completed.

3 Allow time for specifying and suite testing.

4 Schedule known holidays and training.

5 Allow for part-time working if a programmer is concerned with maintenance or other projects.

6 Remember trial turn-round difficulties and try to overlap trials of one program or routine with writing of another program or routine.

7 When dividing time critical programs remember co-ordination and specification overheads.

8 Watch the data preparation and computer loads.

9 When allocating programs consider the need to develop staff experience.

10 Remember that parameters have to be prepared for standard programs and allow time for such tasks.

Summary

The planned use of resources, both people and machine time, is essential to efficient working and successful completion of projects. Without careful scheduling there will be inefficient use of man-power with some staff heavily overworked whilst others are idle, and the machine time requirements will be unpredictable and will tend to fluctuate.

No plan can guarantee success, however success is more likely to be achieved if the plan includes considered contingencies and can be adapted to meet changing circumstances.

Any form of progress control depends on the feedback of actual events against a plan; the plan therefore forms the basis of control.

WORK CONTROL

Control of programming work is necessary in order to ensure that the quality of the work produced is adequate, that end dates are achieved and man-power is used effectively. In order to achieve these objectives programming work must be continuously supervised and progress assessed so that corrective action on sub-standard work, or delays, can be taken as early as possible and wastage of effort can be avoided. These two aspects of control, that is supervisory and progress, are essential activities of any programming supervisor although the amount of supervision will vary according to the experience of the team members, and the degree of progress control will be dependent on the size of the project or installation.

Supervisory control

There are four main objectives of supervisory control:

1 To ensure that the accuracy and quality of all work is of an acceptable level.

2 To ensure that the team functions in an orderly manner, and that staff know their responsibilities, so that effort is not wasted.

3 To ensure that all work produced conforms to the standards in use.

4 To ensure staff receive suitable practical training.

Although formal checking of end products can be used to check the use of standards, accuracy and quality, supervision should also be on a more frequent basis so that rewriting of complete stages is avoided.

Progress control

There are several objectives which have to be considered in any system of progress control:

1 Recording of all estimates which form the basis of the plan.

2 Recording of the plan which forms the basis of control.

3 Checking that all essential tasks are carried out for a project and each program.

4 Comparison of actual progress with expected progress at any stage of a project so that corrective action can be taken on delays as early as possible.

5 Revision of estimates and plan so that progress reporting is realistic.

6 Objective assessment of programmer performance.

7 Collection of statistics so that estimating and control standards may be improved.

The extent to which these objectives can be achieved by a particular installation is dependent on the man-power available for controlling, and the number of people involved in any project.

Method of progress control

A detailed system for controlling progress is given in Appendix 10 and is based on a number of forms on which the various elements concerned in control can be recorded. This system has been designed so that different levels of control requirements can be implemented, and the degree of control can be increased by introducing the next control level.

CONTROL LEVEL 1

The use of three forms gives a very basic work control which allows recording of program estimates and feedback of actual progress against planned, and also provides check lists of tasks. This level of control is suitable for installations where communications tend to be on a day to day basis, and only a few people are concerned with a project.

CONTROL LEVEL 2

Two forms are added to Level 1 in order to introduce formal progress reporting. This level is suitable for installations where more people tend to be concerned in a project and communications need to be on a formal, regular basis.

CONTROL LEVEL 3

Two forms are added to Level 2 to allow the exact recording of actual effort and trial time and therefore provide a more accurate assessment of progress and performance. This level is suitable for installations where very tight control and some statistics are required.

CONTROL LEVEL 4

Two forms are added to Level 3 to allow statistical information to be summarized so that estimating standards can be reviewed and improved. This level is suitable for installations which have staff available for detailed analysis of statistics and updating estimating standards.

Other combinations of the forms can be used, for instance Level 1 plus the two Level 3 forms, according to the needs of an installation.

Summary

Control of work must be exercised if projects are to be implemented economically and efficiently. Supervisory control for product quality, and progress control for achieving the timetable, are equally important. Control is however more than just checking, recording and reporting, since in addition to the assessment of current progress and the taking of corrective action, it must include the prediction of future possible difficulties or delays, and the implementation of procedures to avoid such occurrences. Any installation needs some form of work control, however basic, in order to avoid wastage of resources, excessive costs, and panic measures to complete a project.

TRIALS CONTROL

The trials stage of any program is the one where most control is necessary since there are so many factors which can affect progress. It is therefore necessary to plan in some detail the implementation of the trials stage, the functions and requirements of which have already been designed, as described in Part 2, Chapter 5. This is in order that control can be exercised and the use of resources optimized. The objective of trials control is to ensure that trials are carried out efficiently, economically and within the time limits established for the program.

The degree to which trials are controlled will vary according to the project and installation. Machine costs alone are likely to be a sufficiently large element of the total programming cost of a project to be well worth controlling.

Planning

A plan for each turn-round should be established, that is, which stage of testing is to be carried out and what is to be achieved in each turn-round. This should stay within the estimates of work control (human and computer). Work control only records the number of turn-rounds and computer time taken each week, and man-effort used. Trials control is more on a day to day basis and gives information to work control for assessing the work content that is remaining. Additionally, trials control includes the control of trials submission, release of media, and data preparation, and it ensures orderly testing and avoids waste of computer time. It also ensures that the trials plan is followed and programs are tested fully.

Some of the main considerations when planning trials are:

RESOURCES

1 Staff availability

2 Staff quality

3 Machine time availability

These will generally follow from the overall work control plans.

RESTRICTIONS

1 Target dates

2 Machine time cost

3 Special priorities

4 Special restrictions

These will put constraints, to a certain extent, on planning trials and turn-rounds.

The variations and combinations of these considerations are infinite. Since the prime objectives are to minimize costs and optimize the use of resources then the control of trials will have to be planned carefully and revised in the light of any change.

Technique of control

The simplest and best way of controlling trials is by the use of specially designed forms which allow the recording of events and thus aid in assessing progress. Such records are invaluable for good communication and understanding. The forms also serve as a check list and sometimes can ensure that certain tasks are performed. By monitoring the progress of trials, the programmer and overall performances can be evaluated and action can be taken to avoid delays and waste of computer time. Documentation will also provide a reference and guide for future projects.

For medium and large projects/installations it is useful if the programming supervisor delegates some of his simpler or routine day to day tasks to an experienced programmer. This will leave him more time to carry out other important duties and also assist in training junior staff.

For details of the recommended control forms and their use, see Appendix 11.

Summary of forms

Bar chart	This enables a pictorial record of the estimated and actual progress of testing to be made.
Testing schedule	This aids control over full testing of programs.
Trials submission control	This aids control of the submission of jobs by programmers.
Turn-round	This aids control of the submission of jobs to the computer.
Tape/disc catalogue	This aids control of the use and release of tapes and discs.
Data prep submission	This aids control of the data preparation submissions.
Trials log/analysis	This gives close control over the programmer's individual trials.

Summary

One of the main advantages of positive control of trials is that delays, failures etc. can be recognized immediately so that corrective action can be taken and the consequences resolved. Thus realistic target dates can be met.

By careful control, the wastage of computer time and man-effort can be reduced. Trials will be implemented efficiently and effectively.

An important consequence of these control methods is that less maintenance will be needed.

PART 3 PROGRAM WRITING, ACCEPTANCE AND MAINTENANCE

Chapter 7 Detailed programming

PROGRAMMING STAGES

Detailed programming from the programming point of view is the main stage of a project. Certainly it tends to consume a considerable amount of man effort, and is spread over too long a span of time to be treated as a single stage with a single checkable end product. During this stage therefore each program is considered as a self-contained entity with several sub-stages each of which have checkable end products. This concept of sub-stages is of benefit to any size of installation or project since documentation is produced and checked at various points in the main stage, thus ensuring the accuracy of the work and the availability of complete documentation for maintenance purposes.

The sub-stages

A brief description of the sub-stages is given below and full details are given in the remainder of this section.

LOGIC AND CHECKING

Familiarization with the project

Comprehension of the program specification

Production, or checking, of the outline flowchart

Production of detailed flowcharts

Checking and dry-running detailed flowcharts

CODING AND CHECKING

Coding of the program from the flowcharts

Checking of the coding for logic and syntax

Correcting and re-checking of coding

TRIAL DATA PREPARATION

Preparation of detailed trial data

Calculation of expected results

Conversion of data to appropriate media for trials

Checking of trial data

Preparation of trial running information

Production of draft operating instructions

INITIAL COMPILATION

Production of a program free of language errors, that is, ready for trials

Dry-running of coding when necessary

TRIALS

Processing of trial data

Analyzing trial results and correcting program

Checking corrections

Updating of documentation

Completion of program operating instructions

Advantages of sub-stages

The division of the detailed programming stage into smaller sub-stages allows a higher degree of control to be exercised and also leads to more orderly working.

Training of programmers is made easier since staff are forced to solve problems at various levels rather than trying to start at the coding level; this also tends to improve the design of programs.

The production of the relevant documentation is enforced throughout the detailed programming stage and it is of prime importance that the documentation is kept up to date. Changes due to systems alteration or programming errors must be incorporated in the relevant documents at the time at which they occur since out of date documentation is not only useless but may be positively misleading to other staff concerned with the project.

It is difficult to make programmers realize the importance of staged working and formal document production but these concepts must be enforced since otherwise there is no effective progress control, programs are likely to be inefficient, and maintenance will be an onerous and unnecessarily time consuming task.

PROBLEM COMPREHENSION

It is essential that the programmer fully understands what is required of the program before starting any further work. To facilitate this understanding it is useful to give programmers a knowledge of the purpose of the suite. This background information is included in each program specification but it is also helpful if someone who knows the suite organization, for example, the suite designer, gives a short talk to the programmers describing the project. This is particularly relevant if there are several programmers concerned.

Once having gained an idea of the suite the programmer must study the program specification until all the requirements are understood, asking questions whenever points are not clear. This activity has a dual purpose; firstly to comprehend the problem and secondly to check that the specification is adequate and unambiguous. Obviously in a small installation where one person carries out all the project tasks the problem comprehension is unnecessary but the specification should still be checked carefully because of its use in maintenance.

Method of study

It is not possible to lay down standard methods for understanding the program requirements as each person has their own individual approach to learning; however, it is possible to give some guidance on the study:

1 The programmer should first read the specification to get a general idea of the problem; after that it is a matter of continuously studying the specification raising queries when necessary. It is essential at this stage not to make assumptions but to query any points which are not clear, however trivial they may seem, since only by questioning can it be determined whether the specification is comprehensible and unambiguous.

2 All layouts should be checked for completeness. All records, including heading and end records, should be described and each field should be precisely defined as to sequence, format, radix, length and range of values.

3 The logic of the outline flowchart should be checked carefully.

4 The outline flowchart, list of tasks and processing notes should be cross-checked for consistency and completeness. A check should also be made that the data required to produce the results is available and any redundant items should be queried in case the processing has been misunderstood.

5 The reconciliation and restart requirements, and the error condition actions should be checked for completeness.

6 The trials plan should be studied and used to assess whether or not all the situations have been understood.

All agreed amendments and additions arising from the investigation of the queries must be incorporated into the specification at this stage to ensure that it is a complete and up to date description of the program.

Summary

It is important that programmers are encouraged to raise questions, preferably in written form, so that errors and omissions may be investigated as early as possible. It must be recognized that programmers want to get on with programming and therefore have a tendency to avoid problem comprehension assuming that they will clear up problems as they occur. This must be avoided and programmers must therefore be shown that the earlier they

expose errors the better, since any error may affect many other programs and possibly show up a basic flaw in the system.

FLOWCHARTING

Flowcharts form the design document from which the program is constructed. They are produced in order to help the programmer solve the logical problems without having to think about the detailed coding, and are essential to a maintenance programmer, who may be unfamiliar with the program, in establishing how the program is logically constructed and in isolating the required part of the coding.

Any method of flowcharting must be designed to serve both of these functions, and in addition must allow for changes to be made to the charts with a reasonable degree of ease.

All installations require a programming flowchart standard so that charts drawn by one programmer are intelligible to another, and also to ensure that all charts fulfil the basic requirements as described above. This may seem unnecessary in cases where the writer and maintenance programmer are expected to be the same person, but it must be remembered that staff leave, and even the best programmers forget their own programs very quickly.

Experienced programmers may be capable of coding from a series of scrappy charts but such charts are of no use in maintenance since they are only likely to be understood by their author. Charts which are drawn from the coding do not present a clear picture of the logic, and it is also very rarely that, once the program has been completed, there is time or effort available for documentation. Consequently it is essential that flowcharts are produced and checked before coding begins, and also that they are updated by all changes.

Flowcharting standards

The flowcharting standard, described in Appendix 12, has been designed to include the basic flowcharting requirements.

The method is based on a levelled concept; the outline flowchart (see Chapter 5, *Program structure*, on page 31) forms the highest level of chart and shows the overall logic of the program. Separate charts are then drawn for any box on the outline chart which requires a more detailed solution. This process of expanding boxes to a lower level of charts is continued until at the lowest level the procedure can be easily coded. The number of levels required is dependent on the language to be used and also on the experience of the programmer. Standards are also given for the symbols to be used, box labelling and cross-referencing flowcharts and coding.

Development and checking of flowcharts

The higher level flowcharts, defining the overall logic and structure of the program, will have been produced during the program planning stage and the programmer receives the highest level chart appropriate to the task, that is the outline chart for a whole program or a lower level chart for a self-contained routine.

The programmer takes the given flowchart and produces the detailed flowcharts, level by level, until all procedures can be coded. Generally this means that a box on the lowest level represents five to fifteen instructions or statements, although there are cases where a box generates only one instruction, for example, a test, or a simple sequence of thirty to forty straightforward instructions. Consideration should be given to producing the optimum solution for the main path processing.

The checking of flowcharts is carried out in two stages. Firstly the programmer concerned checks them by using the technique of dry-running. The data prepared during the program specifying stage is suitable for this task since it consists of a list of all the different types of data, and combinations of data, which must be tested. The programmer processes each type and combination of data through the flowcharts to ensure that the detailed logic is correct, in particular file matching and the setting and testing of indicators. In order to do this a copy of the charts should be used, and as the data is processed the reference of the particular piece of data should be noted against each box through which it passes. This will show which paths have been checked and also will help in correcting the charts if errors are found. A separate record should be kept, cross-referenced to the charts and the data, of the state of indicators and data areas.

Secondly the flowcharts must be checked by a more senior programmer to ensure that they perform the specified tasks, that they conform to standards, and have been developed to a reasonable level. The charts should be corrected and rechecked until they are acceptable.

Summary

This method of flowcharting has several advantages:

1 The programmer is encouraged to solve problems in levelled stages.

2 The charts can be developed to the level required by each individual for coding purposes, without making them too detailed for understanding the logic during maintenance since for this purpose the higher level charts required can be selected. (The lowest level charts should not be on a consistent one box to one instruction basis since then they require continual updating.)

3 The levelled flowcharts correspond to the structure of the program and thus the system of flowcharting facilitates the division of a program into relatively self-contained routines either for modular programming or for segmentation.

4 The cross-referencing between flowcharts and coding is of benefit in trials analysis and maintenance.

5 The dry-running and checking of the flowcharts ensures that the logic of the program has been correctly established before the lengthy coding activity is begun.

CODING

Coding forms the basis of the computer program and its production is therefore automatically enforced. There are however many different varieties of coding ranging from the completely unannotated and very complicated to the well-annotated and straightforward, and it is therefore essential to impose some standards on programmers so that a resonably uniform presentation is achieved and maintenance is therefore made easier. It is also necessary to enforce some degree of checking to avoid wastage of computer time and to ensure that the coding is of the required standard.

Coding standards

The coding must relate exactly to the flowcharts and be cross-referenced as described in Appendix 12. Since the logic has already been established on the charts the programmer is free to concentrate on the accurate and efficient use of the programming language. All routines should be written in as straightforward a way as is compatible with reasonably efficient use of the store and time.

All low-level language coding should be annotated with comments which give explanations of the purpose of groups of instructions; high-level language coding need only be annotated where the purpose of a routine is not clear from the statements. Complicated coding must be especially well annotated.

A record of switches and indicators which affect the flow of the program should be kept showing where they are set and used, since this is not always obvious from the flowchart or coding. These records and any other supporting documentation which may be produced to aid understanding should be included either as an appendix to the program specification (see Appendix 6) or filed with the program documentation.

Standards for data-names, program layout, coding format and programming techniques for various languages are given in Appendices 13, 14 and 15.

Checking of coding

Checking of the coding must be carried out by a person other than the author. The responsibility for checking lies with the programming supervisor but the task may be delegated to another experienced programmer.

Checking should ascertain:

1 That the coding is an accurate and complete interpretation of the flowcharts.

2 That the programming language has been used correctly and consistently.

3 That the coding conforms to the standards in use.

4 That the coding is adequately straightforward and annotated for maintenance and comprehension.

5. That the efficiency of the coding is reasonable from the point of view of store space and running time.

To attempt to check all these aspects is difficult and errors may be missed. It is easier and more effective to check through the program more than once, each check covering one or two aspects only.

Corrections arising from any check must be re-checked and it must be ensured that all documentation is updated.

Summary

The use of coding standards improves the legibility of the program and makes it easier to follow.

Checking, although somewhat tedious, is important from the point of view of accuracy and use of standards, and also enables on the job training to be given since areas of improvement can be explained to a junior programmer.

One very important point to be remembered is that a balance between the efficiency of the program and the time taken to write it must be maintained. Although an elegant, highly efficient program may sometimes be necessary, it would not be practical in most cases to allow the writing of such a program to delay an entire project. Moreover this type of program always requires more maintenance effort.

Coding is part of the necessary program documentation and should therefore be written for the use of others as much as for the author.

TRIAL DATA PREPARATION

The programmer needs to prepare detailed trial data and parameters before trials can commence. However, trial data alone is not sufficient preparation for trials. The associated expected results must also be prepared and documented. These will enable quicker and more accurate analyses of trials.

Preparation

The programming supervisor will already have designed the basis of the trial data. This will have been recorded on a testing schedule as outlined in the section *Trials design,* in Chapter 5, page 34.

The programmer may also use the testing schedule forms as an aid in preparing detailed trial data. Each trial data record should have a unique identification for reference purposes.

The overall trial data plan should be checked to fulfil its purpose of completely testing the program. Particular points to note are the limits of items, the different settings of switches, and error paths.

The expected results must be prepared in association with the relevant trial data. With the expected results, trials analysis is simplified and efficiency is increased. This is because a straight comparison can be done rather than having to decide from the results alone whether they are right or not.

Control parameters must also be prepared. These are usually prone to numerous types of error and should be checked thoroughly for both intended and actual content.

Summary

Preparing sufficient and comprehensive trial data and associated results is a very important aspect of trials implementation and should always be considered and allowed for as such. Without good trial data, the quality of a program cannot be evaluated.

INITIAL COMPILATIONS

Initial compilations warrant special consideration in their own right. The objective of initial compilations is to discover and correct all syntactical errors, that is to obtain an error-free compilation listing before trials begin.

Procedure

The parameters for a compilation should be checked thoroughly otherwise the submission may be wasted. There may be certain items of coding which have to be present and valid, otherwise they will cause the compilation to fail. These should be checked.

The compiler usually indicates errors by the use of warning flags. All the errors indicated should be corrected, and additionally logical errors should be looked for.

Summary

An error free compilation will enable the first trial to concentrate on discovering errors of logic otherwise machine time may be wasted in running trials. A compilation listing is probably a better aid to dry-running the program than the coding sheets. Thus it is worthwhile allowing several runs for initial compilations before trials commence.

DRY-RUNNING

Dry-running is a way of testing a program without using the computer. It consists of preparing test data, applying this to the coding, noting the result of each processing step, and checking the results for correctness.

Dry-running is thus a quality control stage. The quality achieved by checking and dry-running involves man-effort but gives a degree of security in predicting the amount and cost of machine time and man-effort for trials.

Machine costs normally form a substantial part of the budget for a project. Consequently, where cost has a higher priority than speed, dry-running should be used. Thus dry-running is essential as a method of progressing program testing with increased efficiency and allowing more effective use to be made of machine time.

Much will depend of course, on the installation. In some rare cases where machine time is free and plentiful the need for dry-running is not so great. Thus dry-running is a cost-benefit decision.

Objectives

Coding should be dry-run to eliminate logical and programming errors which were missed when the full logical check was carried out and which would prove disastrous during trials. The initial aim should be to dry-run the common paths since mistakes in these will delay more valuable trials. All loops and coding containing modifications, counters, setting and resetting of indicators and switches should be dry-run. There is usually a time limit rather than data limit for dry-running but this varies according to the priorities of the project.

Technique

When a program is dry-run ideally there should be a compilation listing and an edit of the trial data which is going to be used for at least some of the trials. Both should have been extensively checked.

Two people who are familiar with the program should be involved in dry-running coding. One person should read the instructions whilst the other person acts as the store by obeying each instruction . The person reading the coding should preferably not be the person who wrote it.

In order to obey instructions, one must record details of, and changes to, all types of store usage, for example work areas, bit indicators, decimal counters.

Summary

Dry-running usually uncovers several errors in a short period of time, any one of which would cause the trial to be abandoned. It also refreshes the mind of the coder as well as giving an understanding of the program to more than one person. This can be valuable if there is a high turnover of programmers. As much of the program as possible should be dry-run using selected records from the trial data prepared for trials. In doing this, the trial data itself will be validated. It is useful to dry-run if trials are held up due to lack of machine time.

Thus dry-running is a valuable task and should be enforced in order to improve efficiency and minimize costs.

It should be considered as a positive part of the testing process which can reduce some of the frustrations of testing on the machine and of running into trivial errors which jeopardize progress.

TRIALS

The objective of running trials is to prove that the finished program is working correctly with the trial data used. In order to achieve this efficiently trials must be planned and controlled. Since there may be a time lag between planning and implementation, some revisions may be desirable in view of any environment changes.

To ensure smooth running of trials, regular file housekeeping and security runs must be performed. Associated maintenance of records of media contents and back-up media cycles must be enforced.

The other particular tasks which make up the trials cycle and which need careful attention are checking procedures and documentation, preparation for trials, and analysis of results.

Checking and documentation

Checking procedures cannot be overemphasised. Carried out properly they can be invaluable in saving effort and time and thus increasing job satisfaction. The latter benefit is one which should be considered all the time and its importance should never be underestimated.

Documentation must be kept up to date to provide an accurate means of communication. If it is not kept up to date then the reliability and usefulness of any recorded information is doubtful. The Trials log/analysis form (see Appendix 11) when correctly filled in provides a good account of trials progress.

Trials preparation

Each programmer is responsible for the preparation of his program for trials, and for the production of any parameters relating to the amendment, compilation and trial of that program.

He must write down on a data form the details which he needs to submit as amendments and parameters. These must be checked by an experienced person before punching. Where possible a stock of standard parameters should be held in order to minimize punching.

Wherever possible all cards should be punched on an interpretive punch and must be checked after punching. If an interpretive punch is not available, and the cards are being hand punched by the programmer, the data should also be punched by a second person and the results visually compared. If an amendment is large and visual checking is not feasible trials should not be run without first checking the listing.

The programmer must prepare his operating instructions and these must be checked by an experienced programmer for factual correctness of intention and detail and must be examined for possible ambiguities of interpretation of any special instructions. Particular care must be taken to ensure that all post mortems required are specified and that tape or disc serial numbers are correct.

Checking is an essential part of programming work. A rotational system of checking is usually impossible to implement; the job of checking should go to whoever in the team is suitably free. Programming errors should be minimized by bringing errors to the attention of the whole team. Stricter checking measures may have to be taken if there are too many of one type of mistake.

Trials analysis

When a trial returns from the machine, the complete results must be studied and not just the reason for the trial failing. It is important to analyze fully the results of a trial so as to be able to make a worthwhile submission and not to rush to submit another trial which may fail for a reason which would have been discovered by more thorough checking.

The programming supervisor should be called upon if any guidance or assistance is needed. In any event, he should spend some time looking at the results of, and amendments resulting from, all effective and ineffective trials. He should ensure that the programmer has fully analyzed the results and has designed the next submission for maximum effectiveness.

Trials analysis should be documented in as much detail as possible by the programmer. He should record the results of the trial, analysis, corrections and comments for the following four reasons:

1 This will provide a comprehensive resumé of the progress of the program which can be useful in the later stages of testing to remind the programmer how and why he did certain things.

2 This will be most valuable if the program has to be taken over by another programmer.

3 The details will be useful in assisting the senior programmer in checking the analysis of the trial and intended amendments.

4 The results and analyses will provide information for statistical purposes which will guide testing on future projects.

Sufficient diagnostics must be incorporated into trials so as to gain maximum benefit from them but great care must be taken not to misuse them. A compromise must be made between the usefulness of a diagnostic and its effect of increasing the trial time (especially printing time) by careful consideration of positioning and frequency. The position should be reviewed after each trial. The objectives are to aid analysis and reduce the number of trials needed.

Summary

The programmer and his supervisor must be satisfied that all the trials will make effective use of the computer.

For each program trial the following tasks should be carried out:

All results should be checked, all errors found and amendments properly conceived.

All amendments should be properly checked.

All parameters to be submitted should be checked both with regard to their intended and their actual contents.

All operating instructions should be checked.

OPERATING INSTRUCTIONS

Operating instructions are the means by which a programmer can inform the operator how to run the program. They must be as full and informative as possible and written in such a way that any operator can follow them easily and as a matter of routine. They are needed as a reminder, even if the programmer is in fact the operator as well.

Contents

Below is listed the most important information required by the operator for running the program:

PROGRAM NAME

The mnemonic name of the program.

PROGRAM DESCRIPTION

The title of the program.

ESTIMATED RUNNING TIME

Total running time of program. The estimated running time will always be given for one-off jobs. For productive jobs, running time is best shown as a function of input volume, for example, minutes per 1,000 cards if punched card input.

CORE

The amount of core storage required.

PRIORITY

The program priority, if applicable.

SOFTWARE ENVIRONMENT

Details of the operating system required.

PERIPHERAL REQUIREMENTS

A list of peripherals required with associated numbers and any necessary details.

MESSAGES

The console messages expected under normal run conditions with associated relevant narrative.

OPERATOR ACTION

All the instructions necessary to run the program. Alternative entry points, selective switch settings, should be indicated. If, for example, long periods of processor activity may be expected, this should be mentioned so that the operator is prepared for it.

Additional information

All exception conditions which may occur during the running of the program should be documented. These are most commonly:

Unusual messages

Program failure

Restart procedures

In addition all program originated messages and any software originated message or machine condition which requires special action must be documented. The type of error or exception conditions giving rise to this message and also the action required of the operator on receipt of this message must also be recorded.

The operator must be informed of the necessary action if the program fails.

If any provisions exist for dumping and restarting the program, the operator must be informed of what action to take:

1 As each point is established.

2 If there is a major interruption to the program.

PROGRAM DOCUMENTATION FILES

The file contains the complete documentation of the program. This documentation is produced and updated throughout the development of the program and therefore little effort is needed to gather this together. It is most important in any size of installation to file all documentation for a program in a logical and standard way as such a file aids maintenance, ensures vital documents are not lost, and also provides useful information for future projects.

Contents

The program file must contain the following documents:

Amendment record

Contents list

Program specification

Supporting documentation

Outline flowchart

Detailed flowcharts

Program listing

Test data (handwritten or computer listings)

Test results (computer listings)

Operating instructions

Control documentation

The updated source program cards or paper tape and the raw test data must also be filed.

Method of production

During the trials stage the programmer must collect the available documentation together and make out an amendment record sheet and the contents list. The program file must be finalized immediately the last program trial has been completed and before the programmer moves on to the next job.

Maintenance

The program file is the basis for good program maintenance, but is only useful if all the documentation is kept updated. An out of date program file can be positively misleading and therefore hinder, rather than help, maintenance programmers.

Once the file has been produced all amendments made should be noted on the amendment record in the form of amendment number, date, description and documents affected.

It is helpful if a formal filing system is set up to house the program files. Such a system should allow all documentation for a project, that is the system and control documents plus the program files, to be filed together. This then forms a good reference library for maintenance purposes and also for guidance on future projects.

Chapter 8 Project acceptance

This chapter describes the final stages of implementing a project, that is, suite testing and final documentation.

SUITE TESTING

There are three main stages to suite testing:

Programmer devised linked trials

Systems trials

Quasi-operational trials

The objective of suite testing is to produce a correctly working suite of programs. The degree of accuracy after program trials is not sufficient in large projects. Large suites need to be comprehensively tested as entities in order to reveal any inconsistencies between the programs. There may be additional inconsistencies in the concept of the system.

The amount of suite testing envisaged for a project must be planned in advance and adequate control procedures must be prepared.

Programmer devised linked trials

The purpose of programmer devised linked trials is to test all possible linkage paths between all programs in the suite. Linking runs can start as soon as the first program in the suite is ready. A program is ready when the programmer has proved it to his own satisfaction and that of the programming supervisor.

The volume of trial data will be small and is specially written to suit the requirements of linked trials. The trial data for individual program trials is not normally suitable for actual linked trials, but may be used to check that any amendments made do not affect the working of the individual programs. The linked trials should be designed to cycle the main file if applicable.

In order to obtain the maximum information from a linked trial it is necessary to process the data through all programs. Data which prevents a program producing an output file should be temporarily removed and the program tested in parallel with the linked trials. Any errors resulting from a linked trial must be fully documented.

Systems devised linked trials

System linked trials commence when the programmer linked trials have been proven 100%. The data is designed and produced by the systems team to test the system and it is their responsibility to see that the final system is correct.

It is imperative that the main files and data files are fully checked after they have been set up in order not to confuse the results of linked trials.

User departments often also provide data for this testing phase, particularly in small installations. In doing so, this will mean that this stage will be merged with the following stage of quasi-operational testing.

The operating procedure guide should be completed and available at this stage.

Quasi-operational trials

There are two stages to quasi-operational trials:

1 Pilot runs

2 Parallel runs

The former are run after linked and systems trials have been 100% successfully completed. The suite is run on a larger volume of data, which should mainly consist of genuine data. The purpose of this stage is to familiarize clerical staff with the handling of data and printed results, to validate any files which may have been set up prior

to operational running, and to familiarize the operators with the job under semi-operational conditions. Any remaining program or systems mistakes should be uncovered by the larger data volume and particular combinations of data.

The latter stage consists of the operational running of the computer system in parallel with the manual system which it is replacing. Results of the two systems are compared. The purpose of this stage is to convince the user that the computer system is performing the correct tasks and functioning perfectly. Parallel running will normally continue for several months.

If the computer system has been set up to do a job which was not done previously, it is necessary to set up a manual simulation of the computer job as a substitute for parallel running. This will also be necessary if the computer job differs from the manual job in major areas, since parallel running loses its effectiveness if direct comparison of results is not possible. Where there is a significant time lag between production of results by the manual and by the computer systems, these results may be compared retrospectively.

Summary

If suite testing is not carried out, then the latent errors will reveal themselves during operational runs almost certainly with disastrous consequences. Even when suite testing has been carried out, the initial operational runs should be vetted very carefully.

OPERATING PROCEDURE GUIDE

An operating procedure guide is necessary in order to provide a complete picture on how to use the system to be implemented. Consequently, it mainly concerns the systems and operations departments. In order to co-ordinate different viewpoints and snags arising from the job, regular meetings should be held between those senior people concerned in the systems, operations and programming departments. The operations department, in particular will have to agree and approve all aspects of operational processing.

The document should contain every relevant piece of information with regards to the organization and running of a suite of programs. Thus it will obviously contain the operating instructions as outlined in Part 3 Chapter 7.

Contents

There are a number of considerations for inclusion in the document:

Miscellaneous information–names and numbers

Background information–details of each program

Operating details

Expected occurrences during run

Detailed data information

Results and other output

Intermediate files

Controls security

Details of any decisions that may have to be made

Details of all typewriter messages

Program failures

Related programs

Preparation

It is desirable that preparation of the guide should begin during an early stage of the project. If they are drafted early and gradually revised more time is allowed for staff to familiarize themselves with the instructions; and, the draft copy of the operating instructions is available for systems trials.

Final production can be left until all specification revisions have been finalized and the draft has been approved by all the interested departments.

If the document is not produced, then the consequences are that people unfamiliar with the system have no immediate and direct source of information if needed. This obviously will cause both delays and errors.

USER PROCEDURE GUIDE

The user procedure guide is the document which describes to the user department the functions which have to be fulfilled, from their point of view, in order to use the system effectively. Although the production of this document is very rarely considered to be a programming function it is advisable that programmers are aware of its existence and contents, since they may have to contribute some of the information.

The document is essential to smooth operational running since it forms the written reference for detailed user department procedures which must be followed in using the system.

Contents

Computer jargon must be avoided, since the guide is for non-computer personnel, and specialized computer terminology will only cause confusion. The more important subjects to be included as as follows:

Objectives of the system

Tasks to be performed, responsibilities and work-flow

Copies of all data forms and detailed instructions on how the entries are to be made

Submission of data, for example timetables, batching instructions

Collection of results, for example timetables, dispersal

Interpretation of results including error reports

Correction of errors and resubmission

Summary

The user guide should be contributed to by the user, system, operating and programming staff, although it is likely that the systems function will have overall responsibility for its production. Without a written reference document the user department may complete forms and submit data incorrectly thus causing many error reports to occur, which they will be unable to interpret or correct. The additional work load of dealing with such a situation will inevitably fall on the computer personnel; it is therefore in their own interest to ensure that a comprehensive and accurate reference guide for the user is available.

Chapter 9 Maintenance

Maintenance is the incorporation of amendments into programs which are already running in an operational environment. These amendments can be due to changes in the system, or to program errors which were not discovered during testing. The amount of maintenance which any project will require will depend to a large extent on how the project was orignally implemented. If formal agreement was not obtained at the appropriate times, or if changes were made without consultation, then there are likely to be many amendments to the system. If the programming tasks were not controlled and checking was ignored then many program errors will occur during operational running.

However orderly the implementation there will be a need for some maintenance, and this should be allowed for when scheduling any new work.

Amendments due to system changes can of course be controlled as a sub-project, and implementation can be scheduled and dates for inclusion agreed. Those due to program error, however, must be made between, or even during operational runs and are therefore more difficult to control.

Responsibilities

Maintenance is normally a joint responsibility of systems and programming, particularly with regard to system changes. Requests for these changes are raised by the users of the system and it is a systems function to establish the requirements and priorities of each change. The effect of the change, the man-effort and the time scale should be agreed formally between systems, programming and the user. The programming work should be carried out with all checking and testing stages included, as in normal programming, and all documentation must be updated.

The correction of program errors should be carried out in the same way as system changes when there is time between operational runs. There are, however, some instances where a correction has to be made before the run can continue. In such cases, more control and checking is necessary, rather than less, since it may not be possible to test the amendment. No change should be made, however urgent, without the correction being thoroughly checked and the results being carefully scrutinized. This checking is likely to be the responsibility of the programming supervisor who must ensure that the amendment will have the desired affect, and that the documentation is updated.

Documentation

The program files contain all the documentation required for maintenance and it is of prime importance that these files are updated to reflect any amendments made. The provision of this documentation allows the maintenance tasks to be allocated more flexibly and avoids total reliance on the original author.

Methods

There are several ways to organize maintenance arrangements from the programming point of view, and these are dependent on the organization for normal project work.

In a situation where projects are implemented by programming teams, a programming supervisor can be made responsible for several operational projects, and any amendments to these projects can be made by members of the current team. The programming supervisor then becomes responsible for allocating maintenance work, and ensuring that it is carried out correctly. This method has the advantage of spreading the load of work over several staff and also avoids relying on a single individual. The disadvantages are that amendments may affect new project work or that implementation of changes may be delayed.

Another approach is to have a separate maintenance team responsible for all operational projects. This method is only likely to be practical in a large installation which can afford the overhead of such a support team. The advantages of using a maintenance team are that amendments can be incorporated without disturbing work on new projects, and that staff with the appropriate skills can be used. The disadvantages are that the team will be unlikely to have an even work-load and there may be times when they cannot cope with all the urgent changes. Staff do not get much job-satisfaction and there may be difficulties of staff turn-over.

A third method is to have one person responsible for all maintenance control and for handing out amendments to whoever is available. This method is flexible in its use of staff but does not ensure any continuity of knowledge of projects, since changes will tend to be given to the original author.

Summary

Maintenance is a function of any computer installation. The amount of effort involved will depend on the way in which the project was originally implemented and the stability of the user organization. Whatever the situation, allowance must be made for some effort to be expended particularly when scheduling new projects. Maintenance work must be controlled very carefully and all changes agreed, checked and tested. The penalty of allowing changes to be made without proper control is that the amendments may themselves be incorrect and the situation may worsen rather than be improved, and thus more man-effort and machine time will be required.

APPENDICES

Appendix 1 Outline suite specification contents

There are thirteen standard content headings for an outline suite specification plus appendices as appropriate. Sufficient information for a resonable degree of understanding should be included in each section together with references to appendices which contain the detailed information, which would interrupt the flow of comprehension if included in the section itself. The headings and details to be included are described below.

1 *Background of the job*

A general description of the organization or department for which the job is to be done, and in particular of those areas and procedures which the job will affect.

2 *The job to be done*

A statement of the principal tasks which constitute the job and an explanation of the need for the tasks.

3 *Data*

A description of each type of data to be provided by the user, or received from other suites. Intermediate data, produced by another program in the same suite, is not included.

4 *Results*

A description of each type of result to be produced by the suite for despatch to the user or other suites. Intermediate results produced by one program to serve as data for another in the suite, are not included.

5 *The job in operation*

A chronological description of the job in operation presented from the user's point of view. The process of organization, checking and despatch of data is described, followed by the receipt and use of results.

6 *Organization for the computer*

A suite organization chart showing each program in the suite and the files which form the communications between programs, including the raw data files, intermediate processing files and the final results. A brief description of the purpose of each program should be included if this cannot be made clear on the chart.

7 *Processing*

This contains details of the processes and calculations required to produce the results from the data provided. These should include:

Reconciliations required

Restart and rerun requirements

Formulae for calculations

Uses of codes and table information

Decision tables

Special actions to be taken for particular conditions

8 *Take-on procedures*

Details of any special procedures that may be required for the creation of master files. This should include original data and results if these differ from those described in the sections on data and results.

9 *Future extensions and enhancements*

Details of any possible extensions to the suite which may be required in the future. Comments on any enhancements which could be made to the design of the suite if circumstances change.

10 *Initial implementation estimates*

Estimates of man-effort and costs for all further activities and a timetable showing project completion date and resources required. Alternative timetables should be included if the resources available are likely to vary. Details must be given of all factors taken into account in making these estimates.

11 *Preparation and running times*

A summary of the times when results are required and when data must be supplied, and estimates of the computer time required to run each program. The data volumes assumed in calculating the running times must be included, together with any variations which are likely to occur.

12 *Preparation and running costs*

A summary of the costs of data preparation, computer time and off-line procedures, which will be required for operational running. These should be based on the volumes shown in 11 above, and should show ranges of costs due to variations in data volumes.

13 *Glossary of terms*

An explanation of any terms used which are unlikely to be self-explanatory.

14 *Appendices*

These include:

File and print layouts

Specimens of pre-printed data forms

Specimens of pre-printed results stationery

Examples of printed results

Details of subjects referred to in other sections

Appendix 2 Systems specification vetting

In order to vet or interpret the systems specification the programmer must be able to understand any conventions or formal disciplines used. This appendix introduces the systems specification forms recommended for use by ICL.

There are three forms to describe data and results:

File description

List of file items

Derivation of file items

There is one form to summarize documents and their contents:

Output analysis chart

There are two forms to describe interactions between data:

Processes form

Decision table

DATA DESCRIPTION

File description

The purpose of the form is to give an overall description of a file. This form will be supported by one or more sheets describing in detail the records and the items which make up the file.

File is used in its widest connotation to mean any collection of associated records. This form is used to describe both existing files and the files required by the proposed system.

VERSION

All files which contain the same information but whose records are in a different sequence are regarded as different versions of the same file. For example, invoicing movements which are sorted successively to customer number for updating the debtors' file, to commodity code for updating stocks and to area code for sales statistics, would have the same file code with a different suffix.

The use of this system obviates the necessity to repeat detailed descriptions of the file.

SEQUENCE PRODUCED

This describes the order in which data is held on the file. This may be in the form:

IMMATERIAL	where the sequence is of no relevance.
RANDOM	where the file is in true random order.
ONE ONLY	where only one item is held on the file.
..WITHIN..ORDER	where the file is sequenced item within item order (for example, employee number within department code order).

MEDIUM

The medium on which the file is held: ledger cards, magnetic tape, etc.

ICL

File description

Ref.no. W

Organisation

Project name	Project Code
Identity	Code

Version	Sequence produced	Medium

Type	Priority	No. of copies
Size of file Min	Max	*Av / Abs	Received from/sent to*
Frequency produced	Retention	
Condition		

*Delete as appropriate

FORM C14/107/7(11.69)

TYPE

Here the file should be specified as being an input, output or reference file. (A reference file is one which is updated in the system or otherwise referred to.)

PRIORITY

The priority for the output files should be inserted here. For example, before job *x* or 1600 Wednesday.

NUMBER OF COPIES

This is used primarily for computer files output to a printer.

SIZE OF FILE

The size of the file is normally expressed as the number of records it contains. Where this is variable the minimum, maximum and average sizes are stated. Where it is fixed it is stated as the absolute number of records.

RECEIVED FROM/SENT TO

The source of the data by department or function name is entered here in the case of input files. For output files the title or function of the recipients of each copy is noted.

FREQUENCY PRODUCED

For an output file it is necessary to specify the required frequency of production, for example, weekly, daily.

RETENTION PERIOD

This is used to specify the period for which the file is to be retained as current.

CONDITION

The condition which gives rise to the production of results is stated. The statement generally takes the form FOR EACH... WHEN: for example, for each customer's account when a matching order has been received.

List of file items

This form describes in detail the records which constitute the file. A separate form is made out for each type of record comprising the file.

ITEM CODE NO.

A unique code number may be given to each item or group to facilitate future reference.

IDENTIFIER

Here is inserted the name of the record or item or item group.

LEVEL

A numerical value is used to show the hierarchical relationship of items within the form.

PICTURE

In this column the format of the data comprising the item is given in COBOL notation.

OCCURRENCES

Where any item or item group is repeated within a record, an entry is made to define the number of occurrences. Where this is variable it is stated as a minimum, maximum and average number. Where it is fixed the appropriate figure is entered as the absolute number.

Y REF. (ITEM DERIVATION FORM REFERENCE)

Where it is required to add further descriptive information, for example the calculations required to arrive at the value of the item, the reference number of the form on which this information appears is entered here.

ICL

List of file items

Ref.no. X

Organisation

Project	Project code
Identity	Code

Item code no.	Identifier	Level	Picture	Occurrences Min.	Max.	Av/Abs*	Y Ref	Notes

*Delete as appropriate

FORM C14/107/8(11.69)

ICL

Derivation of file items

Ref. no. Y

Organisation

Project | Project code

Identity | Code

Item code no.	Origin or method of calculation	Sequence & condition	Associated items: Item code	Associated items: File code	Associated items: X Ref	Notes

FORM C14/107/9(11.69)

Page of

Derivation of file items

This form is used to give any additional information that may be necessary about any item or item group.

ITEM CODE

Here is specified the reference code of the item or item group to be described.

ORIGIN OR METHOD OF CALCULATION

The means of arriving at the value of an item is described here. For example, the selection of a price to be charged for a particular commodity from a number of prices on a master commodity may be determined by a digit of the purchaser's code number indicating whether he is a wholesaler or a retailer.

SEQUENCE AND CONDITION

If an item group can appear several times within a record, the order in which successive groups are arranged is stated. If an item appears only in certain circumstances this is stated.

ASSOCIATED ITEMS

The code numbers of the record or item and file containing information used in the calculation of the time described may be included, together with the reference number of the list of file items form on which that item is described.

OUTPUT DESCRIPTION

The output analysis chart shows all output documents and details of the fields to be included in each document.

The names of the documents are listed on the left hand side of the chart together with a note of their frequency. All elements of data that appear on any report are listed across the top of the chart, together with their sizes.

The codes, which are entered where the rows and columns intersect, show which information appears on which report, and now the field is derived:

R for information entered on the report from records.

S for information derived from source documents.

C for information that is the result of calculation within the system. (The calculations necessary to obtain these fields will be specified on either a derivation of file items form or a processes form.)

PROCESSING DESCRIPTION

It may be seen that the data and output description forms not only define the individual items of data but also they define simple interactions between items of data to produce others. Major processing actions are however described separately.

Processing may be described in one of two ways: as a conventional sequential description or as a decision table. The method used may depend both on the originator and on the application. There is certainly a move towards using decision tables as their use can offer many advantages. For this reason decision tables are described separately in some detail in Appendix 3.

The form used will depend on the method being used, (see pages 68 and 69) but whichever form is used the description should be quite unambiguous. The descriptions will utilize the unique references from the data description and standard mathematical and logical operators.

SUMMARY

The systems specification forms are designed to present in a clear and unambiguous manner those facts required by the suite designer:

All input, by type and times of availability

All main, or standing file items

All tasks (interaction between input data and main file data to produce results)

All results, by type and times required

OUTPUT ANALYSIS CHART

COMPANY Flag Publishing Co. DATE 16 Mar. 1970

PROJECT 60/A/22 COMPILED BY R.J. Robinson

Data Fields

*N = Numeric
A = Alpha
A/N = Alpha/Numeric

Frequency			Customer No.	Traveller	Branch	Name & Addr (Invoice)	Name & Addr Consignee	Date	Order No.	Code No.	Item/Description	Qty Ordered	Price	Extensions	Purchase Tax	Cash Disc	Gross	Net	Credit Limit	Current A/c Bal	Month A/c Details	Trav Sales Hmly Value	Branch Sales " "
	Field Type, A, N or A/N*		N	N	N	A/N	A/N	N	N	N	A	N	N	N	N	N	N	N	N	N	N	N	N
	Field Size in Characters	Maximum	5	2	3	86	80	6	8	6	36	6	6	8	5	5	9	9	5	9	80	9	10
		Average	5	2	3	58	50	6	5	4	12	4	4	5	4	4	6	6	5	6	36	6	7
	Output Documents																						
D	INVOICES		R	R	R	R	S	C	S	R	R	S	R	C	C	C	C	C					
D	CREDIT CONTROL REPORTS		R	R	R			C											R	R	R		
D	A/C QUERIES		R	R	R	R		C											R	R	R		
M	STATEMENTS		R	R	R	R		C												R	R		
M	TRADE REPORTS (A)				R			C		R		R											
M	CREDIT STANDING REPORTS				R			C												R			
RFQ	PRICE CHANGES AND OTHER NOTICES		R	R	R	R		C				LETTER FORM NARRATIVE PLUS NEW PRICES, CHANGE OF P/T ETC											
M	TRADE REPORTS (B)		R	R	R			C							C		C						
M	COMMISSION TAB			R	R					R		R	R									C	C
Q	QUARTERLY ABSTRACT				R																	C	C

R = Record S = Source C = Calculated

ICL

Processes

Ref. no. P

Organisation

System

Type

Referred to in

No.	Function	Description

FORM C14/107/5(11.69)

ICL

Decision table

Ref.no. T

Project | Reference | Author | Date

Table name

Number of conditions | Number of actions | Number of rules

		Rule numbers																
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	Else
		Relative frequency (%):																
No.	Conditions																	
No.	Actions																	

FORM C14/107/10(11.69)

Page of

Appendix 3 Decision tables

Decision tables provide a concise and easily understandable method of documenting problems which, in many cases, is superior to using flowcharts or written accounts. Their main advantages are:

1 They are more readable and hence easier to comprehend because

(a) A distinction is made between conditions and actions.

(b) Operations are specified in parallel rather than serially.

2 When constructing decision tables, there is no need to specify a solution or partial solution by implying an order for the testing of conditions.

3 Inconsistencies are easy to detect and remedy.

4 Amendments are easy to make because all affected areas are readily apparent.

Decision tables are particularly suitable where a problem involves different processing according to several conditions but are still applicable, though no more efficient than other methods, to cases where actions are applied serially without intervening conditions.

STRUCTURE OF DECISION TABLES

A decision table consists of four basic elements:

1 The condition stub (or condition statement)

2 The condition entry

3 The action stub (or action statement)

4 The action entry

The first two elements specify the value of the conditions relevant to the problem. The last two elements signify the actions to be performed when certain conditions hold. These four elements are arranged in a table with the four quadrants separated by double horizontal and vertical lines.

Condition stub	Condition entry
Action stub	Action entry

The two halves, action and condition, are linked by rules which run vertically between the condition entry and action entry quadrants. This will be clearer from the following example which defines the action to be taken on receipt of a request for an article from a store.

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
Article in stock?	Y	Y	Y	Y	N	N	N	N
Article on order?	Y	Y	N	N	Y	Y	N	N
Permission to issue article obtained?	Y	N	Y	N	Y	N	Y	N
Apply for permission to issue article		X		X		X		
Issue article	X		X					
Reserve article		X		X	X	X		
Reject request							X	X

The condition entry represents replies to the questions in the condition stub where Y represents yes and N represents no. An X in the action entry signifies that the corresponding action in the action stub is to be performed. Therefore, if the following conditions exist:

Article in stock?	Yes
Article on order?	No
Permission to issue article obtained?	No

the condition entry is inspected to find a corresponding rule. In this case rule 4 applies and on following this rule to the action entry the required action can be found by referring to the action stub and entry, that is apply for permission to issue article and reserve it.

From the example it can be seen that rules 7 and 8 lead to the same action with only the third condition (Permission to issue article obtained?) differing. The third condition is irrelevant to the action and the two rules can be combined. A dash (–) is placed in the condition entry for these two rules to signify that the condition is not relevant to the rule. Similarly rules 1 and 3, and 2 and 4 can be combined resulting in the following table:-

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5
Article in stock?	Y	Y	N	N	N
Article on order?	–	–	Y	Y	N
Permission to issue article obtained?	Y	N	Y	N	–
Apply for permission to issue article		X		X	
Issue article	X				
Reserve article		X	X	X	
Reject request					X

In the following example, a subroutine ZA, ZB or ZC is performed if a condition A, B or C respectively is true. Otherwise, a subroutine ZERROR is performed.

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7
Condition A is true	Y	Y	Y	N	N	N	N
Condition B is true	Y	N	N	Y	Y	N	N
Condition C is true	–	Y	N	Y	N	Y	N
Perform subroutine ZA			X				
Perform subroutine ZB					X		
Perform subroutine ZC						X	
Perform subroutine ZERROR	X	X		X			X

From the example, it may be seen that four rules result in the same action, and these rules cannot be combined. This is not economic, especially in cases with more conditions, so a general rule, known as an ELSE rule, is introduced. The ELSE rule is used when no other rules apply and is normally placed after the other rules. Therefore the above example using the ELSE rule would appear as follows:

	Rule 1	Rule 2	Rule 3	ELSE
Condition A is true	Y	N	N	
Condition B is true	N	Y	N	
Condition C is true	N	N	Y	
Perform subroutine ZA	X			
Perform subroutine ZB		X		
Perform subroutine ZC			X	
Perform subroutine ZERROR				X

So far the order of the conditions and actions within the condition and action stubs has been immaterial. In practice the order in which actions are performed is often important. For example, a record must be edited before it is printed.

Therefore a method of signifying which actions are to be performed first is needed. One method is to assume that actions are performed in the order they are listed in the action entry. Thus in the first example permission to issue the article would be applied for before reserving it. This method has two disadvantages.

1 An order for the actions is specified even though it might not matter in some cases.

2 It is possible that two actions are performed in one order under one rule and a different order under another. This means that one or more actions must be duplicated in the action stub.

Another method, which does not have these disadvantages, is to replace each X in the action entry by a number. Then the first action to be performed for each rule is signified by '1', the second by '2', the third by '3' and so on. When the order of two or more rules is of no significance they are given the same number.

For example, a rule performs actions A, B, C, D, E and F, and the following constraints apply:

B must be performed before C or D

C and D must be performed before A or F

A and F must be performed before E

They would then be marked in the action entry as follows:

Perform action A	3
Perform action B	1
Perform action C	2
Perform action D	2
Perform action E	4
Perform action F	3

The decision tables which have been considered so far are known as limited entry decision tables and are characterized by specifying all the conditions and actions in the condition and action stubs. The condition entry contains only the answers to the conditions in the condition stub. The answers may be

Y for yes; the condition is true

N for no; the condition is false

-the condition is not relevant

The action entry contains an indication of whether an action is required for prevailing conditions. The action may be given as

X for action required

1,2,3 etc. for action required in the specified order

No entry is made if an action is not required

Another type of decision table, known as extended decision tables contains only part of the conditions and actions in the stub, the rest being in the entry part of the decision table:

Example

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	ELSE
Record type =	A	A	A	A	B	B	B	
Total : 100	<	<	⩾	⩾	<	⩾	⩾	
Adjustment type	= 10	>30	⩽10	= 25	-	= 45	= 35	
Perform subroutine	ZA	ZB	ZC	ZA	ZC	ZA	ZD	ZERROR
Write record to	File 1	File 1	File 2	File 1	File 2	File 2	File 2	

Each condition is evaluated by combining the two portions in the stub and entry to give true or false. A rule (except the ELSE rule) is performed when all conditions, excluding irrelevant conditions, are true. Each action is obtained by combining the action stub and action entry for the particular rule

From the above example:

1 if the record type is A, the total is 150 and the adjustment type is 25, then subroutine ZA is performed and the record is written to file 1 (rule 4).

2 if the record type is A, the total is 73 and the adjustment type is 15, then subroutine ZERROR is performed (ELSE rule).

The third type of decision table, mixed entry decision tables, is a combination of limited and extended entry tables.

Example

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	ELSE
Calculate indicator set?	Y	Y	Y	Y	N	
Number of figures =	1	2	3	4	0	
Set P =	A①	A+B①	A-B①	A-B①		
Set Z =	0①	0①	C①	C+D①		
Calculate $R=P^2-Q^2$		X②		X②		
Calculate $R=P^2+Q^2$	X②		X②			
Set R=0				X①		
Set R=999						X①

Notice that because of the extended entry form it is no longer possible to signify the order of actions by the use of a number for the action entry. Instead, though it is not necessary in this example, the order number is placed in a circle in the top right hand corner of each relevant box.

CONSTRUCTING AND CHECKING DECISION TABLES

Decision tables are ideally suited to a systematic and logical procedure for their construction which will usually save considerable time and effort in practice.

Limited entry decision tables

The conditions and actions that are relevant to the particular decision table are selected. Operations which occur at different logical levels (for example, file handling and individual record processing) should not occur in the same decision table. Methods of linking decision tables are discussed later.

The condition and action stubs are then completed.

Example

A bonus is paid according to sex, age and skill. The possible bonuses are 5%, 10% and 15%.

The table as constructed so far is:

Male?	
Skilled?	
Age≥25?	
5% bonus	
10% bonus	
15% bonus	

The next stage is to complete the condition entry quadrant. This can be done by adopting the following technique.

The maximum number of rules for any decision table 2^n

where

n is the number of conditions.

In this example $n = 3$ so there are 8 rules.

The first condition entry is completed by inserting Y's against the first condition for half the rules and N's for the other half. The number of Y's = $\frac{2^n}{2}$

Male?	Y	Y	Y	Y	N	N	N	N

The next condition entry is completed by inserting Y's under half the Y's and half the N's in the previous condition entry. N's are inserted in the remaining spaces.

Male?	Y	Y	Y	Y	N	N	N	N
Skilled?	Y	Y	N	N	Y	Y	N	N

This continues until the condition entry is complete.

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
Male?	Y	Y	Y	Y	N	N	N	N
Skilled?	Y	Y	N	N	Y	Y	N	N
Age ≥ 25?	Y	N	Y	N	Y	N	Y	N
5% bonus								
10% bonus								
15% bonus								

Considering each rule in turn the action entries can be completed. This will reveal any impossible conditions. For example, if for some peculiar reason it is impossible to have an unskilled man over 25 then rule 3 above can be deleted. However, it is usually better to incorporate impossible conditions in an ELSE rule with error action. Assuming that none of the rules is impossible the situation might now be:

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
Male?	Y	Y	Y	Y	N	N	N	N
Skilled?	Y	Y	N	N	Y	Y	N	N
Age ⩾ 25?	Y	N	Y	N	Y	N	Y	N
5% bonus			X				X	X
10% bonus				X		X		
15% bonus	X	X			X			

The table is now complete and the problem is to simplify it. The following condition must be observed when combining two rules:

Two rules can only be combined if their actions are the same and condition entries are identical except that for one, and only one condition, one rule contains a Y and the other an N.

Example

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
Condition 1	Y	Y	Y	Y	N	N	N	N
Condition 2	Y	Y	N	N	Y	Y	N	N
Condition 3	Y	N	Y	N	Y	N	Y	N
Action 1	X	X	X		X	X	X	
Action 2				X				X

In this case rules 1 and 2 can be combined because they have the same action and same condition entries except that rule 1 contains a Y for condition 3 and rule 2 an N.

Similarly rules 3 and 7, rules 4 and 8, and rules 5 and 6 can be combined resulting in:

	Rule 1	Rule 2	Rule 3	Rule 4
Condition 1	Y	–	N	–
Condition 2	Y	N	Y	N
Condition 3	–	N	–	Y
Action 1	X		X	X
Action 2		X		

The resultant rules show that rules 1 and 3 have the same action and also the same condition entries except for condition 1 where rule 1 has a Y and rule 3 an N. Therefore these two rules can be combined resulting in the following table:

	Rule 1	Rule 2	Rule 3
Condition 1	–	–	–
Condition 2	Y	N	N
Condition 3	–	N	Y
Action 1	X		X
Action 2		X	

Besides reducing the table to a simpler form this procedure also reveals redundant conditions. Condition 1 contains only dashes suggesting that the condition is irrelevant for each rule and can thus be omitted.

In the original example the following simplications can be made:

	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6
Male?	Y	Y	Y	N	N	N
Skilled?	Y	N	N	Y	Y	N
Age ⩾ 25?	–	Y	N	Y	N	–
5% bonus		X				X
10% bonus			X		X	
15% bonus	X			X		

Extended and mixed entry tables

Extended and mixed entry tables are different in that the condition entry quadrant cannot be completed in such a systematic way. The condition and action stubs are completed in the same way as for limited entry decision tables. The relevant values for the first condition are then noted.

Example

Record type =	A | B

The value of the first condition is considered and the relevant value is applied to the second condition. The number of rules is increased as necessary.

Record type =	A	A	B	B
total : 100	<	⩾	=	≠

Conditions are further applied until all condition entries have been completed.

Record type =	A	A	A	A	B	B	B
total : 100	<	<	⩾	⩾	=	≠	≠
Adjustment type	=10	>30	⩽10	=25	–	=45	=35

If at any stage a condition does not apply then a dash (-) is inserted and the number of rules is not increased.

Because all possible rules are not usually included it is advisable to use the ELSE rule to signify the action for these unspecified rules.

Having constructed a decision table it should be checked for the following types of errors.

Contradictions

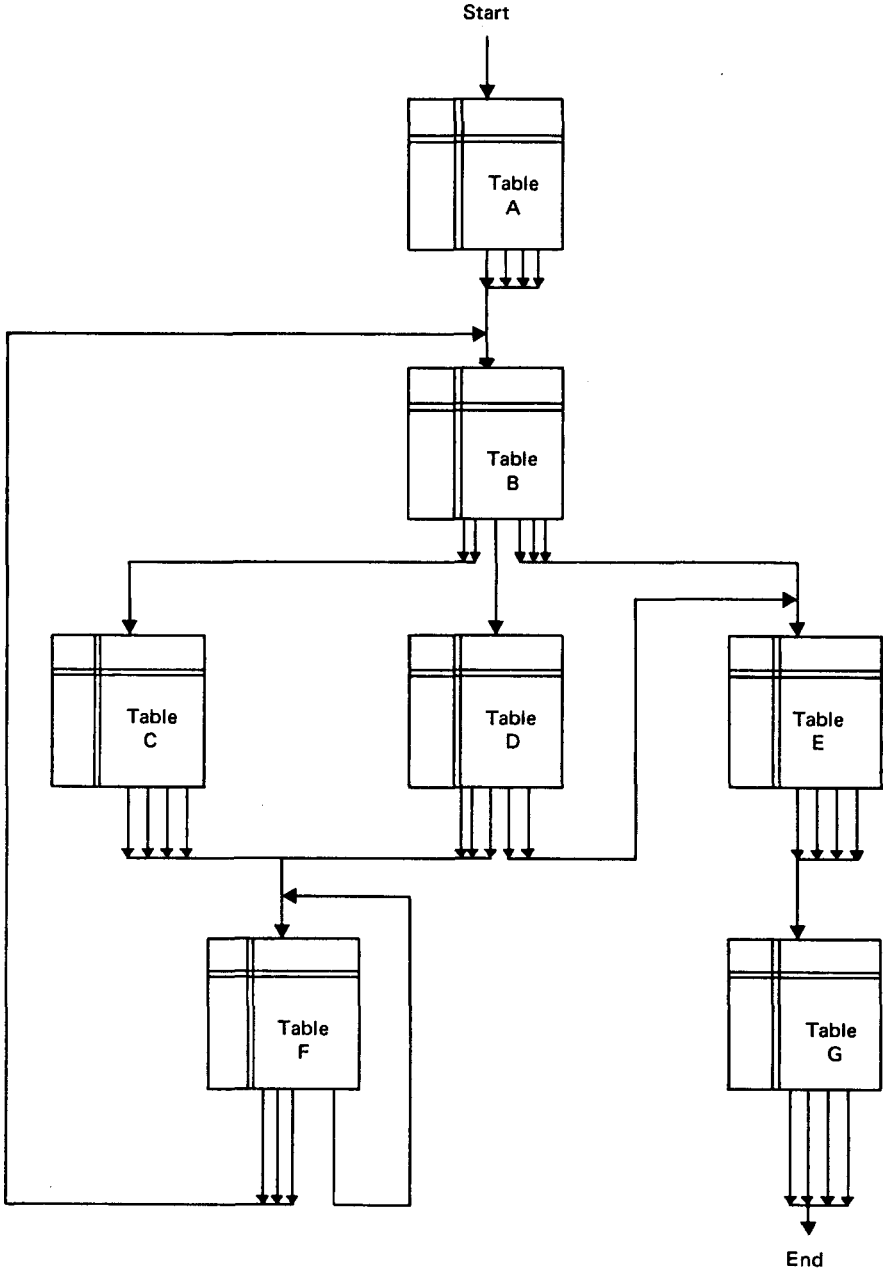
Contradictions occur when two or more rules are satisfied by the same set of conditions. This cannot occur for limited entry tables if they are constructed as suggested. For extended and mixed entry tables, this type of error can only be discovered by inspection.

Redundancies

Redundancies in the action half of a table occur when actions are specified in the stub but no rule specifies that this action is to be taken. This can easily be detected by inspection. Redundancies in the conditions of limited entry tables are revealed on simplification of the table; any condition with all dashes in the entry quadrant is redundant. Extended and mixed entry tables must be inspected for this type of error.

Completeness

Decision tables with an ELSE rule are automically complete, but may not be correct. Limited entry tables without an ELSE rule can be checked for completeness by comparing the number of rules with the known maximum. Extended and mixed entry tables without an ELSE rule can only be checked for completeness by inspection.


Start
Table A
Table B
Table C
Table D
Table E
Table F
Table G
End

Accuracy

The stub entries should be checked for accuracy, particularly to ensure that limited entry tables conditions admit exclusively to a yes or no answer and that the question is phrased correctly.

APPLICATION OF DECISION TABLES TO PROGRAMMING

There are two different approaches to using decision tables for program specifications. The first uses flowcharts and a written account to specify most of the program, using decision tables only for particularly intricate processing details. The second uses decision tables to specify the majority of the program and a written account for details which do not fit into the decision table structure, such as error message layouts. Whichever method is adopted it is almost certain that all the information cannot be held in one decision table and therefore some method of linking decision tables together is required.

The first essential is that every decision table is given a name and this is usually placed above the condition stub.

Tablename	Rule numbers
Condition stub	Condition entry
Action stub	Action entry

The control can now be transferred to another decision table by making the last action of a table

GO TO *tablename*

where *tablename* is the name of the decision table containing the required conditions and actions.

The control does not return to the first decision table (unless of course the second table includes this action). It is imperative, then, that the transfer action is the last action of a table, or subsequent actions will not be performed.

Decision tables can be linked sequentially as shown in the diagram on page 78.

It is perfectly feasible (as in table F) to re-enter the same table though it is presumed that the actions performed the first time will cause another rule to be performed and will eventually lead to an exit from this table.

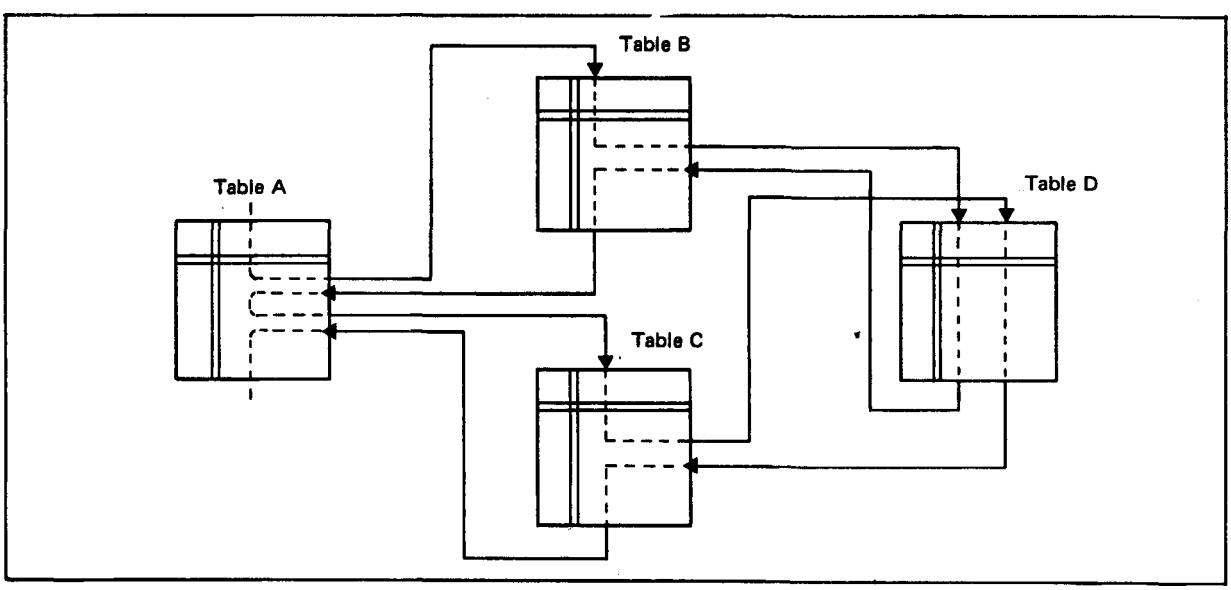
The second method of linking decision tables is to incorporate in the action stub of the first the action:

PERFORM *tablename*

where *tablename* is the name of the decision table containing the required conditions and actions. The last action of this decision table will be

EXIT

and the control will return to the action immediately following 'PERFORM *tablename*', in the first decision table.



These two methods can be used together, but it should be remembered that control is always transferred to the first action of a decision table.

When a programmer is presented with a program specification using decision tables the tables must then be converted to coding. One way is first to convert the decision tables to flowcharts and from these write the coding. This in part defeats the advantages of decision tables, and is not really recommended. Undoubtedly the best method is to use a pre-processor which accepts information in a decision table format and converts it to coding, usually in a high level language ready for compilation. At the present time this is somewhat wasteful on storage due to inefficiencies in the pre-processors but will certainly improve in the future. For medium size programs this is still probably the best method of obtaining coding because of the decrease in writing and testing time.

If a pre-processor is not available then coding can be obtained directly from the decision tables either by setting up the condition as a multi-way switch which branches to the action parts for each rule or by using a decision table analyzer routine.

Various articles have been written on methods of obtaining efficient multi-way switches. The following articles may be profitably studied.

Egler, J.F. A procedure for converting logic table conditions into an efficient sequence of test instructions, *Communications of the ACM,* Vol. 6, No.8, Sept. 1963, p 510-514.

Montalbano, M. Eglers procedure refuted, Letter to the Editor, *Communications of the ACM,* Vol. 7, No.1, Jan 1964, p 1.

Pollack, S.L. Conversion of limited entry decision tables to computer programs, *Communications of the ACM,* Vol. 8, No.11, Nov. 1965, p 677-682.

Press, L.I. Conversion of decision tables to computer programs, *Communications of the ACM,* Vol. 8, No.6, June 1965, p 385-390.

Sprague, V.G. On storage space of decision tables, Letter to the Editor, *Communications of the ACM,* Vol. 9, No.6, May 1966, p 319-320.

When a decision table analyzer routine, also known as a rule mask technique, is used, the parameters are set according to the state of each condition, and the routine will cause the appropriate action to be taken.

Further details of this subject may be found in the following articles.

Barnard, T.J. A new rule mask techniques for interpreting decision tables, British Computer Soc., *Computer Bulletin,* May 1969, p 153-154.

King, P.J.H. Conversion of decision tables to computer programs by rule mask techniques, *Communications of the ACM,* Vol. 9, No.11, Nov. 1966, p 796-801.

Kirk, H.W. Use of decision tables in computer programming, *Communications of the ACM,* Vol. 8, No.1, Jan. 1965, p 41-43.

Veinott, C.G. Programming decision tables in FORTRAN, COBOL or Algol, *Communications of the ACM,* Vol. 9, No.1, Jan. 1966, p 31-35.

Veinott, C.G. More on programming decision tables, Letter to the Editor, *Communications of the ACM,* Vol. 9, No.7, July 1966, p 485.

SUMMARY

Decision tables provide an extremely powerful aid to program specifying because of their simplicity and wide application. Even if they are used for no other purpose they are important and should be considered very seriously. As pre-processors improve in efficiency they will become even more important. Already they are acknowledged to reduce writing time considerably and, because of their format, fewer errors occur, thus reducing testing time in spite of the extra time needed for compilation.

Appendix 4 Data security

This appendix introduces the principles and economics of data security in the design and planning stages. Data security is concerned with maintaining the integrity of all the data flowing through the system.

The primary concern of data security is the recovery from errors and the detection and prevention of errors.

ERROR RECOVERY

Error recovery is defined as the combined programming and operator action required to maintain logically continuous input/output operations in spite of equipment malfunction. This section discusses error recovery at three levels:

1 System level recovery

2 Program level recovery

3 Device level recovery

System level recovery

This consists of the procedures required to recover from any loss of data which may result from any equipment malfunction. Such contingencies must be considered at the design stage. The procedures involved mostly fall into the categories of file reconstruction and back-up facilities.

Program level recovery

The primary objective at this level is to minimize the loss of processing time caused by any permanent error. This might involve, for example, periodic dumping of intermediate results from a program to permit it to be resumed at a point immediately prior to the occurrence of the error. Techniques exclusively concerning random access devices at this level are discussed in the appropriate direct access manual.

Device level recovery

This is the standard hardware/software response to transient input/output errors. Device level error recovery on System 4 and 1900 Series machines is handled by hardware facilities and executive routines.

TYPES OF ERROR

The principal sources of error are:

1 Hardware (including media)

2 Software

3 Operator

4 Data

5 User program

Hardware errors

There are two types of hardware error:

1 Transient hardware errors

2 Permanent hardware errors

Transient hardware errors on standard devices are normally handled by the standard software such as the reissuing of an unsuccessful input/output operation or calling for brief standard operator intervention. This type of error does not persist after recovery measures have been taken. The errors which persist after recovery measures have been exhausted are considered to be permanent errors.

Permanent errors fall within the scope of program level and system level error recovery. Examples of permanent errors and tape breaks, printer ribbon tears; that is, situations requiring extensive manual recovery procedures. Recovery from a permanent input/output error depends to some extent on hardware considerations. Recovery also depends upon system constraints (imposed at the system design stage) which are not hardware-dependent. An example of such a constraint is the re-run dump cycle in a tape system. Re-run dumps are taken at predetermined intervals (at end of reel of output, every six minutes, every 10,000 records, etc.). The length of the intervals represents the maximum amount of reprocessing that the user is willing to accept in case the program has to be suspended.

Software errors

These errors are unpredictable, so preventive action cannot be taken. However, provision must be made for the detection and recovery from these errors. Prevailing conditions largely determine to what extent a particular software error is acceptable.

Operator errors

It must always be accepted that no manual procedure is entirely reliable, and for that reason systems should normally be designed to minimize operator intervention. To minimize errors in necessary intervention, operator action should be standardized as far as possible for any installation. In the early stages of system design the user and the systems designer should collaborate to produce a set of operating standards. Tried and proved techniques should be standardized in all installations where possible.

Data errors

Errors in primary input data should be detected by normal data vetting. Under no circumstances should errors in such data be permitted to cause unscheduled interruptions in the operation of a system. Such an interruption will, therefore, be deemed a program error, caused either by faulty programming or inadequate systems design. Errors can, of course, occur in system control data and where software checks are inadequate to detect these at the earliest possible time, system checks should be incorporated. For example, additional file control procedures might be specified to check the user's own file labels.

User program errors

Such errors, due either to inadequately designed or tested systems, are unfortunately probably the largest source of error other than transient errors which are automatically corrected by executive routines when they occur. The most important conclusion to be drawn from this is that since these errors will occur, plans must be laid in advance for dealing with these contingencies.

ERROR DETECTION

This section discusses error detection at three levels:

1 Systems level detection

2 Program level detection

3 Device level detection

Systems level detection

This may take the form of special programs which perform periodic checks on the integrity of the data. For example, it is not uncommon for accounting suites to accumulate totals both daily and weekly or weekly and monthly, these results being compared periodically for reconciliation checks. For files updated in situ, reconciliation programs can be run periodically to accumulate and check totals over the total extent of the file. As the file is updated in situ it is unlikely that any other function will ever access the whole file in one run. Further, with non-exchangeable random access media it may be prudent on large fixed files to accumulate details of detected errors. An analysis of the occurrence of these can give advance warning of failing or lower reliability than normal, to enable re-organization or relocation of the file to be effected in advance of actual failure.

Program level detection

This section is restricted to a summary of the more important techniques and certain adaptations specifically applicable to files on random access media. Technical details of input/output error detection and file protection are covered in Appendix 5. There are two main techniques for program level error detection:

1 Input/output record counts

2 Reconciliation totals

Both these techniques need to be adapted for random access files and other files updated in situ, as they normally rely on reading every record on the file. One of the advantages of random access media is the ability to access only records active on that run. For this reason total records are held on the file containing record counts and totals of strategic amounts, and on updating runs running totals of insertions, deletions and amended amounts are updated on the file. Periodically run reconciliation programs access the whole file, accumulating new file totals and reconciling them with the old totals and the running totals on the file. The new totals are included and the running totals cleared. It will often be convenient to include a reconciliation check in a file re-organization program.

Device level detection

Parity checking takes place on input and output. Various technical format checks are performed which are handled by executive routines. Cyclic redundancy characters are generated on output and regenerated and checked on input, and a READ AFTER WRITE CHECK may be performed without retransferring to store. On 1900 Series machines this check is standard.

On System 4 computers the check is optional. The check will obviously increase processing time and channel usage. While this check is necessary for the earliest possible detection of errors on large files on fixed disc, for small files it may be found that the expense incurred by the check is greater than the cost of an occasional reconstruction run.

FILE PROTECTION AND RECONSTRUCTION

These are, respectively, the procedures involved in preventing data loss or corruption, and reconstituting data in the event of a loss occurring. These constitute the important consideration of data security. Since file protection can never be entirely relied upon, file reconstruction is considered first.

File reconstruction

A different approach is required for the two principal types of file organization:

1 Sequential organization (including serial files)

2 Random organization (including indexed sequential files which are not processed as strict sequential files)

SEQUENTIAL ORGANIZATION

In sequential files at any point of time during the processing of a master file one can consider the file as consisting of two distinct parts: one part being completely processed and the other being completely unprocessed. If this point of division is identified by noting the physical point of alignment of the master file(s) before abandoning the run of a program for any reason, the program can be recommenced from this point. By periodically noting this point of alignment throughout the processing of a file, any section of the processing can be repeated in isolation.

RANDOM ORGANIZATION

With the random updating of a file on a direct access device the problem is quite different. While the logical classification of records into updated and not updated still exists, the file can no longer be divided into two distinct physical areas corresponding to these. Further, as the processing of a single record might be spread out in stages over the entire course of updating, the sharp distinction of updated and not updated records is not clear. Thus it may be necessary to devise a system whereby each individual act of updating is a separate restart point, perhaps by recording on each record as it is updated a unique serial number identifying and corresponding to each updating action. A further complication is that a file might be accessed by more than one program concurrently. If the sequence of updating is significant it will not be possible to attain exactly the same sequence again by re-running the programs due to the complex hardware/software interaction. One solution to this problem is to write all updating records (or copies of updated records) to a log file in the sequence in which they are applied. Processing can then be repeated by a program reading this file instead of the original input data.

Back-up

Back-up is the name applied to hardware, data and system support available to an installation for use when one or more pieces of equipment necessary for normal operation of any system becomes inoperable for any significant period of time. The importance of considering back-up in systems design cannot be stressed too strongly. However reliable any piece of equipment may be, it is bound to fail some time, and although the average frequency of such failure can be predicted, the individual occurrence cannot. It is the systems designer's function to ensure that the ensuing disruption of service is minimized.

HARDWARE BACK-UP

This can be illustrated by considering a magnetic tape oriented installation. It is often advisable to have at least one tape deck more than is actually required for the running of any system, either by providing one deck more than the maximum requirements or by designing all applications to run with one deck less than the total complement. Then in the event of failure of one tape deck, sufficient back-up facilities will exist to allow processing to continue.

However, the user and the systems designer must establish an alternative procedure in case a breakdown should occur in a critical piece of equipment. This may consist of the negotiation to have the system run on another machine with a similar configuration or it may consist of no more than awaiting the repair. However simple the procedure, it must be established before a failure occurs.

DATA BACK-UP

Data back-up is an insurance against actual loss of data due to equipment malfunction, program failure, or accident. The type of data back-up feasible depends largely upon the sizes of the master files.

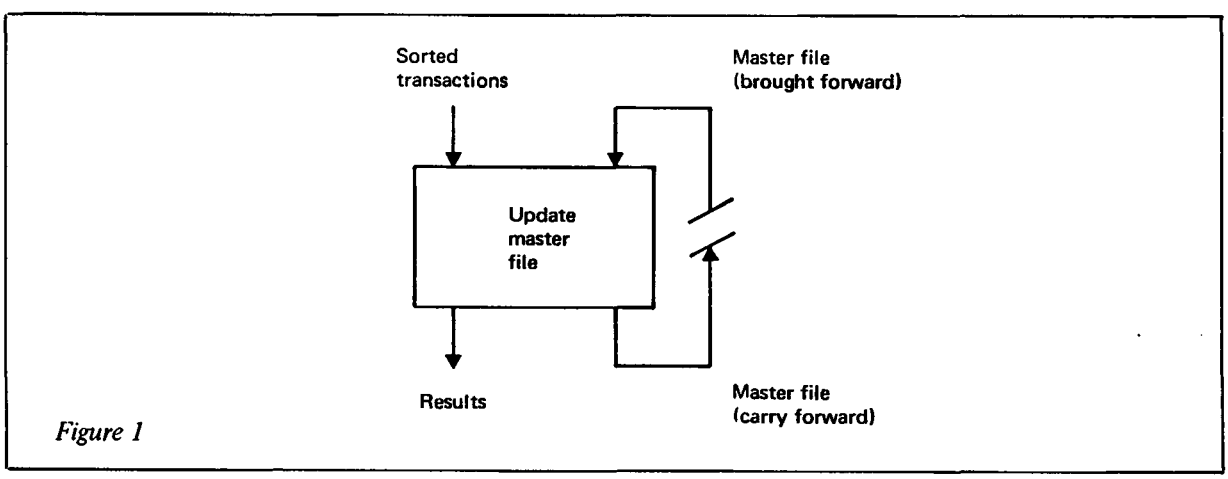


Figure 1

Figure 1 illustrates a simple brought forward/carry forward updating system with separate files. The well-tried reconstruction method used here is to retain the input data file and the old brought forward version of the master file. By retaining the input to two or more cycles, called the Grandfather, Father, Son system, the latest version of the master file can be reconstructed at any time. The degree of security is in proportion to the number of cycles retained. The value of this must be off-set against the additional cost of the storage medium and the off-line storage room.

A particular aspect of this system is that for files held on random access media such retention is wasted unless the different versions of the file are held on different data packs, otherwise a single accident could destroy the total reserve of data back-up.

Figure 2 represents a system perhaps more frequently used on random access media. This is an updating system where the master file is updated in situ. The brought forward file is destroyed by the action of updating, but a copy of this file can be retained. The frequency of making this copy will depend upon the size of the file, the length of reconstruction run acceptable, and the degree of security required. For example, if a file is copied before every six runs of the updating program, the cost of producing the copy will be about one sixth the cost of copying it every run, but the length of a reconstruction run, if necessary, will be six times as great. With really large files it may well be impractical to copy the whole file, and various approaches may be made. It may be satisfactory to copy a portion of the file prior to each run, covering the whole file in a cycle. Alternatively, the best approach may be to make a back-up copy of the file and subsequently keep a copy of each record updated. Periodically the updated records could be merged with the master copy.

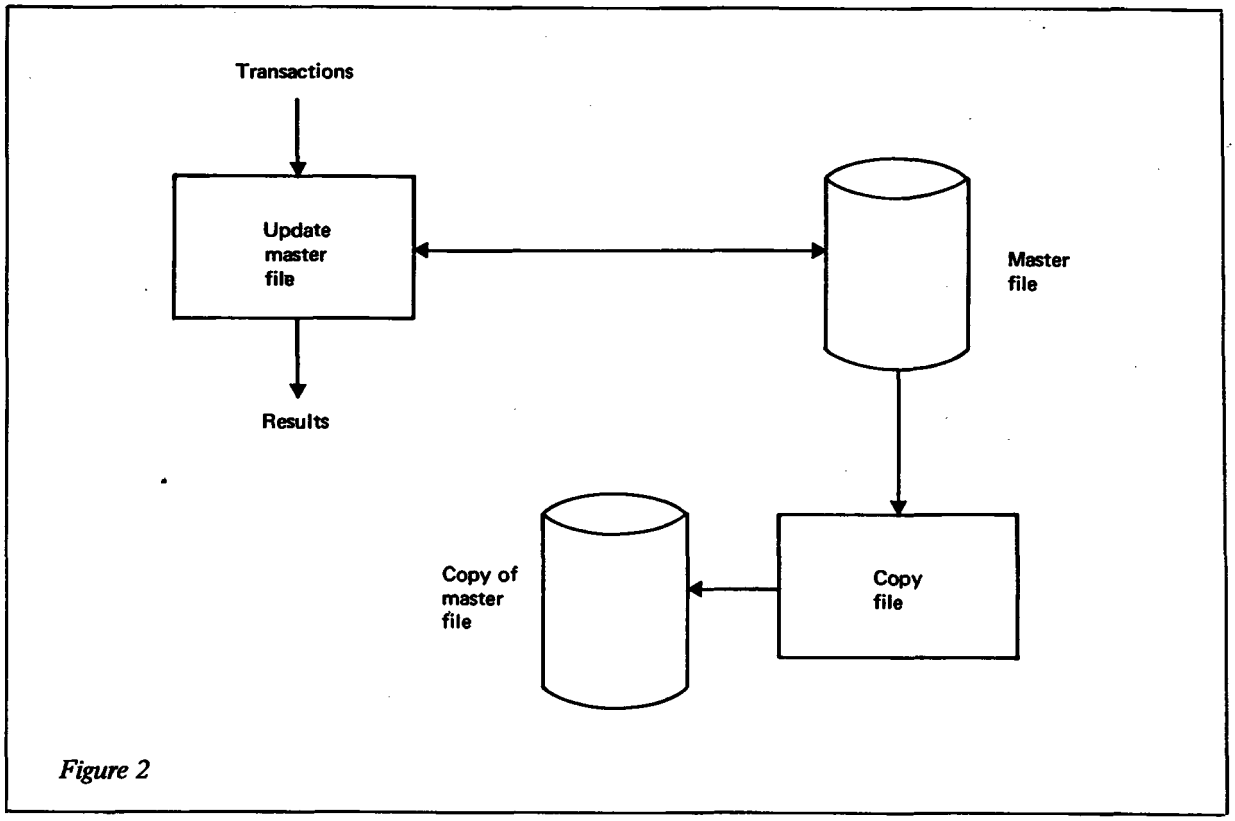


Figure 2

Copying a main file usually provides an ideal opportunity for file re-organization and restructuring. It is also a good opportunity to carry out overall reconciliation procedures which cannot be done during random processing.

SYSTEMS BACK-UP

Figure 3 illustrates an example of systems back-up, bypass and reconstruction procedures on magnetic tape of a random-oriented file which is normally updated on fixed disc. A system maintaining a large file on fixed disc may require that operating continues even when the fixed disc is inoperable. If so, an alternative processing scheme to permit the system to continue functioning must be designed. At the worst the bypass system could allow the system to continue, handling only the more important processing. Alternatively it might be possible for the bypass system to handle the full work load at the expense of other work.

Summary of reconstruction techniques

File reconstruction techniques should be established at the inception of the system design stage. The reconstruction techniques should include an allowance for:

1 read or write errors on a random access device. This procedure should form a part of the reconstruction package

2 inability to copy a random access file to a back-up device because of hardware failure

3 inability to copy a back-up file to random access because a hardware failure caused faulty recording of the back-up file

The frequency with which off-line back-up files are checked depends on the devices available to the system and on whether whole or part files are copied.

File protection

This can be achieved by the interaction of system, software and hardware factors.

System protection will take the form of the planning of intelligent file allocation between devices and disc packs. Special care must be taken to maintain the safety of operational files on fixed disc from corruption by operator error or program testing failure.

Software protection includes the comprehensive label checking which is a standard part of the executive systems. Further checks are built into the file handling software preventing any program error from rendering a file unreadable.

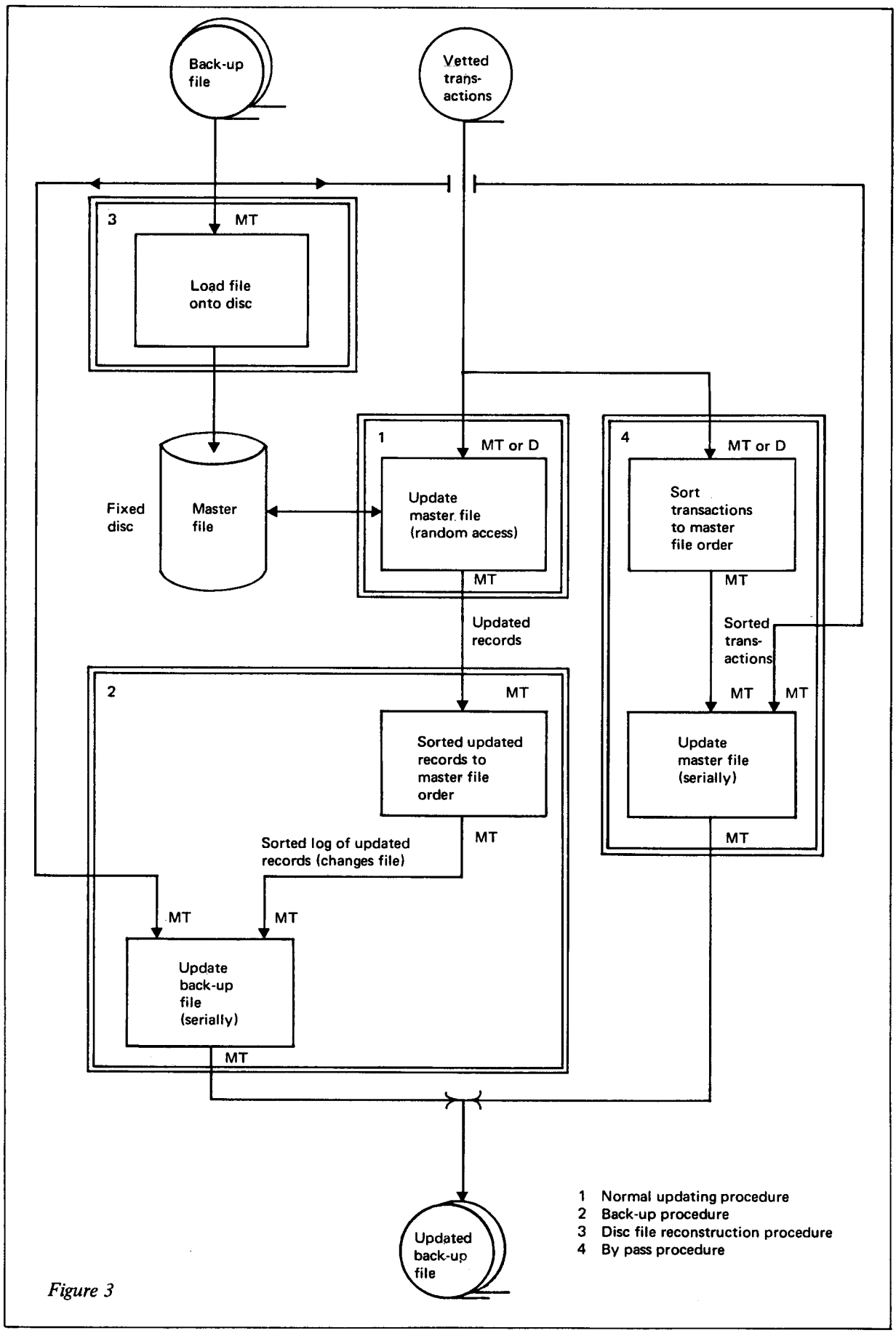

Back-up file
Vetted trans-actions
3
MT
Load file onto disc
Fixed disc
Master file
1
MT or D
Update master file (random access)
MT
4
MT or D
Sort transactions to master file order
MT
Updated records
Sorted trans-actions
MT
MT
2
MT
Sorted updated records to master file order
MT
Update master file (serially)
MT
Sorted log of updated records (changes file)
MT
MT
Update back-up file (serially)
MT
Updated back-up file
1 Normal updating procedure
2 Back-up procedure
3 Disc file reconstruction procedure
4 By pass procedure

Figure 3

Hardware protection takes the form of copying files. An advantage of direct access devices over magnetic tapes or cards is that since the recording surfaces are not touched, a file will not deteriorate with repeated reading. Hence unlike other media, standing files need not be periodically recopied.

File protection is discussed more fully in Appendix 5.

ECONOMICS OF DATA SECURITY

No generalized rules can be proposed to govern the selection of data security procedures, but this section summarizes the factors which will influence the decision.

The overriding factor that determines the selection of a specific system is cost. The systems analyst must evaluate each method in terms of money, because each method requires the use of computer time (reducing through-put) and additional off-line storage media (requiring additional capital investment). The ultimate selection must be justified in terms of the amount of computer time that would be wasted if no recovery mechanism is included in the system, in terms of maximizing the amount of productive computer time available, as opposed to alternative methods, and finally, the selection must be justified in terms of the service which the computer installation provides the organization.

Immediate cost of error recovery

ADDITIONAL PROGRAMMING

Error recovery procedures require more programming, coding, testing, and documentation time. Sometimes the additional programming can be separated into independent programs and subroutines included in processing programs. The more these subroutines accomplish, the simpler and faster the independent programs will be; the converse is also generally true. In deciding which should bear the burden of the work, the planner should investigate whether the subroutine can be packaged and used by all programs.

ADDITIONAL CORE STORAGE

The more work each recovery subroutine performs, the more core will be required for each program. This can affect the ability to multiprogram. The additional core storage requirement must be balanced against its impact on the size of the independent recovery programs.

INCREASED PROCESSING TIME

The additional processing required per record to effect error-preventive techniques increases the total processing time per record. This increase must be balanced against the amount of processing time required for the independent program.

ADDITIONAL OFF-LINE STORAGE

Each error recovery scheme should be evaluated in terms of its requirements for off-line storage media.

Note: As magnetic tape is much cheaper per character stored than disc packs, a magnetic tape deck is often justified in a configuration for the sole purpose of back-up copying.

ADDITIONAL DEVICES

Each error recovery scheme must be evaluated in terms of the number and kind of additional devices required per program. For example, the use of a log tape requires the allocation of at least one additional tape deck to a program. Additional device requirements also affect the ability to multiprogram, and may even cause program restructuring because of lack of devices on-line.

Continuing cost of error recovery

REDUCTION OF THROUGHPUT

Any error recovery scheme requires a percentage of the total computer resource. The planner's function is to keep this percentage down to a minimum, as well as to maximize the amount of processing per record.

OPTIMUM LENGTH OF THE RECONSTRUCTION CYCLE

The length of the reconstruction cycle is in proportion to the process of reconstruction. However, a short reconstruction cycle must be run frequently. The optimum balance must be established in terms of the device being used.

DESIRED LENGTH OF RECONSTRUCTION DATA

The planner must evaluate the cost of having to maintain several generations of reconstruction data against the cost of having to reconstruct data manually.

Other factors affecting error recovery

MULTIPROGRAMMING

On systems with few input/output devices and a small amount of core store, multiprogramming may affect the error recovery procedures incorporated into each program, as these increase the program sizes. Independent error recovery programs will not increase the general program sizes.

PROGRAM STREAMING

Because programs within streams may pass files to their successors within the stream, the analyst should investigate the possibility of using a common log tape. This is possible when a program stream consists of programs accessing the same direct access file, and is usually possible even when the programs access different direct access files. Thus, when log tapes are merged to produce input to the reconstruction cycle, fewer tapes are involved.

SHARED FILES

When two or more programs running concurrently may access the same disc file, the error recovery scheme must be designed to identify the sequence in which each program updated the same record. One technique that may be employed is to attach the date and time of day to the image of the updated record before writing it to a log tape.

SUMMARY

Incorporating error recovery into system design must be done in a manner that covers all possible error conditions, with a minimum encroachment on productive computer resources. This encroachment must be minimum for both short-term and long-term operations. The error conditions which the recovery scheme encompasses must provide against the loss of data resulting from:

Acts of God (fire, floods, earthquakes etc.)

Equipment malfunction

Program error

Operator error

Appendix 5 File design

The term file design may cover many topics, from the basic record/block structure of a magnetic type file, through the efficient design of a conventional master file, to the specification of a data base. The latter is not covered in this appendix since the aim is to introduce the fundamental considerations and to make the case for the construction of files on the basis of logical design rather than historic accident. While this appendix lays out the fundamentals of direct access file design no attempt is made to duplicate the wealth of detail in the direct access manuals, to which a direct access file designer should refer. Here are discussed the decisions to be made by a suite designer in designing files and details some of the possible options. There is no attempt, however, to give that basic knowledge required of anyone working on this subject, that is, a thorough understanding of the machine and software facilities being used. To design a file to give the most efficient service possible the designer must know the file handling features of the programming language to be used.

A file should be designed with two principal objectives in mind:

1 To minimize running time, usually by minimizing peripheral transfer time and attaining maximum concurrency of peripheral transfer with processing

2 To maximize the use of the available resources

These two aims are usually mutually incompatible and the art of file design is to choose the most acceptable compromise, based on a knowledge of the volumes of data involved.

Further care must be taken to achieve consistency across the whole suite of programs. A file should not effect an economy in one program at the cost of great inefficiency in all the others. The frequency with which the file is used in the various programs constituting the system must be considered.

THE CONSIDERATIONS OF FILE DESIGN

The basic considerations of file design are:

1 The selection of an appropriate medium

2 The position, relationship and format of each data item

3 The ability to exercise control over the file

4 The ability to reconstruct the file

Choice of medium

Two types of file are considered here: those which communicate transient facts between programs, and those which maintain the basic data of the system. In either case the choice of medium usually becomes a choice between magnetic tape files and disc files.

The mistake is sometimes made of thinking of random access storage as superseding more conventional storage media. Every medium has uses to which it is particularly suited, and all the peripherals available to the designer should be considered on their merits.

The processing of magnetic tape, because of its physical nature, requires sequential batch processing. The times involved in realigning and random searching are prohibitive in a magnetic tape file. Also it is not possible to alter selectively a single block in the middle of the tape, as the tolerances on the tape transport are not sufficiently small to allow the consistent packing density necessary to predict accurately the physical length of any block. To change anything on a magnetic tape file the whole file normally has to be rewritten, from the point of change onwards.

On random access devices three facilities alter the approach to the organization of processing:

1 The read/write heads can traverse the whole extent of the device in a very short time (on most devices this time is less than 150 milliseconds)

2 Each block of data can be directly addressed and accessed by the hardware.

3 The constant rotational speed is sufficiently accurate to allow the overwriting of individual blocks

These factors allow significantly different approaches to be made. For example, inactive records on a sequentially organized file need not be read. Also records can be updated is situ, that is, they can be read, processed and written back to the same point on the disc. This way, only records which are to be updated need to be written. A less obvious advantage of random access devices is the ability to update several master files with one pass of the transactions, as the number of files per device is no longer limited to one.

The advantages of direct access processing are not without a cost. Sequential storage with indices may require mutliple accesses to locate a record and incomplete packing of the storage medium to accommodate additions and extensions. Back-up and restart problems are magnified by the fact that when a record is altered on a random access device, the old record is lost forever. There is no grandfather master file for retrieving records from a file of an earlier generation as in tape processing.

Position and relationship of data items

The first stage is to determine, in detail, all the information which will be required on each particular file. The design of the file then depends on the determination of the position of each item of data in the file, relative to the position of the other items of data. The items should be formed into records, and the layout within each record decided upon. Where appropriate, the most suitable format (binary, decimal, character) for each field should be determined. Any additional control records should be specified at this stage, and the sequence of records determined. These records are then grouped into blocks, to give the complete file layout.

File control

Each file should be designed to permit strict control of its use. It should only be possible for a file to be used by those programs for which the file is intended, and then only on the appropriate run(s) of the program. The control of files is described in the appropriate reference manuals.

File reconstruction

All files must be capable of being reconstructed in case of loss, or damage in any way. For a serial file, reconstruction takes place from a previous generation. The previous generation is reprocessed against the appropriate current data to produce the required file. Alternatively the master file may be copied in each run, and the copy stored separately. However, if the original file was corrupted before or during the copying, then the copy will also be unusable and it will still be necessary to make use of the previous generation. The reconstruction of files is described in Appendix 4.

The following factors must also be considered for files on direct access media.

File activity

This is most often defined as the number of different records accessed per run divided by the total number of records in a file.

The percentage of activity is one of the factors to be considered. If a low percentage of the records are to be processed on a run, the file should probably be organized in such a way that any record can be quickly located without having to search through all the records in the file.

The distribution of the activity is also a consideration. With some methods of organization, some records can be located more quickly than others. The records processed most frequently should certainly be the ones that can be located most quickly.

Given conditions of file activity and average hits per access, there exists a break-even point where the total processing time in a serial system is equivalent in time to processing in a random access system. Just where this lies for any given application depends on many factors, but normally it occurs in a file where the activity is less than 5%.

Inquiry time value

A data processing system which batches transactions or inquiries produces information of diminishing value. The value diminishes in direct relation to the batching period. The limit of decreasing the batching is reached with real-time systems. Generally the running cost increases in inverse proportion to batching time, so the value of the results must be balanced against the cost.

Volatility

This term refers to the addition and deletion of records to a file. A static file is one that has a low percentage of additions and deletions while a volatile file is one that has a high rate of additions and deletions. No matter how the file is organized, additions and deletions are of significant concern and can be handled more efficiently with some organization concepts than with others.

Size

A file that is too large to be on-line at one time must be organized and processed in certain ways. A file may be so small that the method of organization makes little difference, since the time required to process it is very short no matter how it is organized.

The growth potential of a file must also be considered. Usually, files are planned on the basis of their anticipated growth over a period of time. Initial planning must also consider how growth that exceeds this size will eventually be handled.

Summary

In order to achieve efficient file design, the designer must choose the device, by defining the system for each device or combination of devices, rather than by trying to adapt one system to cater for all possible combinations; and define the layout of all items within the file, considering the provision of adequate file control and file reconstruction.

BASIC STRUCTURE OF A FILE

The principal considerations of structuring a file are as follows:

The best sequence of records

The necessity for additional information, such as control totals

The access method for direct access devices

The choice of an identifying key for a record

The most effective record layout and size

The manner of holding data within the record

The activity of the file

The block size

The file length

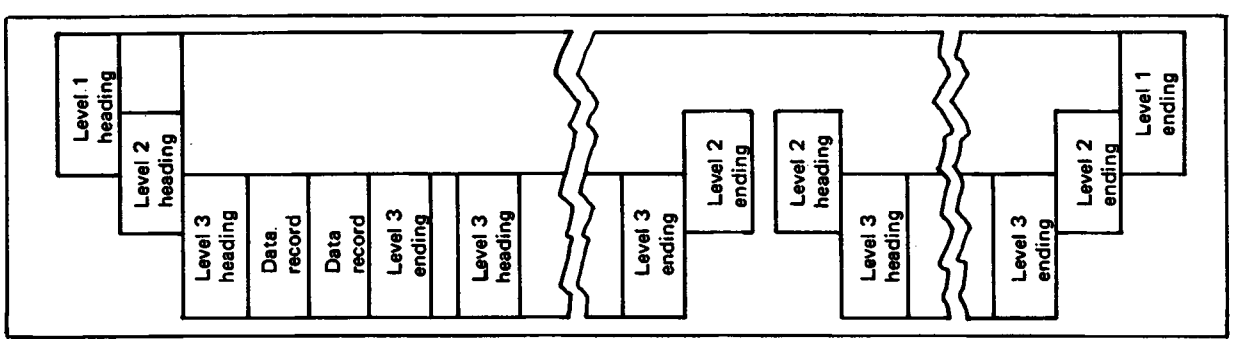
Method of approach

The designer will normally use his experience to choose a particular file structure as a basis of the design. He may make a number of initial standard assumptions and certain overall systems aspects (particular control records, sequences, etc.) may need to be considered as constraints.

Depending on the accuracy of his initial assessment (which in turn will depend on his experience) the designer may then have to consider each of the factors listed above and modify the design in the light of this consideration.

General structure

Most serial files have the following structure:



Files may have records at any number of levels, not necessarily three as in the illustration. It is the task of the planner to determine the number of levels which the file will require, and where the information being considered is to fit in a structure of this type. A simple example may help to explain how information may be positioned in a file to give a structure as illustrated.

EXAMPLE OF FILE STRUCTURING

A company has a number of factories in each of which there are several paying points, that is, places where the employees are paid their wages. Wages for all employees are calculated in a single job run on the central computer installation. The job includes a file, updated weekly, with the above structure.

The details of the records chosen are as follows:

Level 1 heading record: the company heading record

This record contains information governing the file, and information pertinent to all items within the file. There are three types of information:

1 To identify the file

2 To set up the store prior to processing

3 To be used in calculations applying to all employees, regardless of factory or paying point, (income, tax rates, etc.)

Level 2 heading record: the factory heading record

Each of these records contains two types of information:

1 To identify the factory

2 To be used in calculations applying to all employees within the factory, regardless of the paying point

Level 3 heading record: the paying point heading record

This record contains two types of information, similar to the factory heading record:

1 To identify the paying point

2 To be used in calculations applying to all employees within the paying point

Data records (within level 3)

There is one record for each employee, containing his name and key number, and running totals of his pay, tax, allowances and deductions for the tax year to date.

Levels 3, 2 and 1 ending records

These are the ending records for the paying point, factory and company respectively. These are the last records for the relevant group and contain reconciliation totals for the group just processed, and running totals for the file to date.

If the sequence of records had not already been specified, the program planner would at this stage assume that the records in each level are in ascending order (alphabetic or numeric) within the level above. Thus, in the example above, employees' records are in ascending key number sequence within ascending paying point number sequence within ascending factory number sequence. One data record was chosen for each employee, although of course two or more records may be used, depending on the size of record decided upon and the volume of information it is necessary to hold for each employee.

Control records

There are two types of control record: those based on the file structure, and those which are independent of the structure. The heading and ending records at each level of the file structure can be considered as control records

of the first type. The heading record at the highest level should be used as a file heading record, and as such it should contain information relating to the file as a whole.

Each level heading record controls all records following it until a further heading record of a higher or equivalent level is encountered. Similarly, the ending records contain totals which represent the state of the file and of the group just processed at the appropriate level. Thus the ending record at the highest level will provide overall totals for the whole file.

Another type of control record, probably occurring immediately before or after a level ending record, is any record necessary for providing restart and re-run facilities.

Sometimes it is necessary to include in a file information which does not fit into the type of structure already described. Information which it is difficult to fit in satisfactorily is, for example, a table of data which has to be updated at the end of one run from the information collected during processing so that it can be used at the beginning of the next run for processing. If the file is on a direct access device, no problems should arise. However on magnetic tape if the table is held at the head of the file it is impossible to update it once this point is passed. It would be necessary to write the new table at the end of the file, but this still means that at the start of the next run the file has to be run forward to the table, the table read, and the file rewound before processing can begin. A better solution is to write the table onto another file. If another magnetic tape deck is not available, but results for printing are being written on a tape, the table could be written there instead. Cards or paper tape could also be used.

Direct access file organization

File organization can be defined as the relationship of the control fields of the file to the physical location of that record in the storage medium.

Although a file of records can be arranged in a storage medium in many different ways, the storage techniques can be classified as either sequential or random.

Sequential file organization implies that adjacent records in a file are sequenced alphanumerically or numerically, in ascending or descending order. Particular fields, located in the same relative positions within the data records, are selected as sort control fields for a specific file sequence. These are the keys.

A random file organization implies that records are stored without regard to the sequence of their record control fields.

Sequential and random file organization does not necessarily imply sequential or random processing. The terms sequential and random, when used in conjunction with processing, usually refer to the order of the input transaction records or to the order of the reference to records in the master file.

One fundamental consideration when organizing information on any medium is the method by which the information can be located when it is desired to retrieve it. The control fields by which records are usually identified are known as keys. There are several methods of locating information in a file according to a given key.

On magnetic tape this problem is solved by holding records in a serial key order. Provided processing is in serial order it is then quite feasible to search the tape, examining each record until the required key is found, since it is known that the required key will be held further down the tape than the last processed record.

On direct access devices each track can be individually addressed, and data may be accessed in any specified track. Records can be addressed in the sequence of their hardware addresses as for magnetic tape, or individual blocks can be accessed at random.

File access

The manner in which a file is accessed need not bear a direct relation to its organization.

The methods usually employed are:

1 Strict serial: This means that all processing must be done serially, and updating must, in general, produce a new copy of the information because of the difficulty of handling insertions. Furthermore, because the address associated with a specified key is not known, inactive records must be searched.

2 Indexed random: In this method a key index is held enabling the address on the disc corresponding to each key to be found. The appropriate track can then be located and searched for the record. The file need not be held in a sequential order on the disc. However, this method has the disadvantage that large indexes are required and thus index search time may be high.

3 List processing: Records in a file organized for list processing are characterized by the use of pointers. Pointers may be held to indicate the address (or addresses) of the next record in the sequence. Usually the file is organized as described under random organization, for random processing. When the file is processed sequentially, each record supplies the address of the next record in sequence. This technique of the file organization eliminates the need for an index for sequential processing. Record insertion under this technique is simplified. For example, if a file has two records, A and C, A initially points to C. When record B is to be inserted, B is written into any available area; A is retrieved, its pointer to C is moved to B, and the A pointer is modified to point to B. Records may be deleted using this method.

List processing provides the facility of integrating several related files on the same random access storage device, since each record may contain more than one pointer. For example, an inventory file may consist of records describing the quantities of finished products on hand, one record per product type. Each record may contain pointers to records in other files, as for instance, a record in a bill of materials file describing the parts requirements for a single product of that product type. Records in the bill of materials file, in turn, may contain pointers to sub-assembly and parts inventories.

While this method has the disadvantage of often requiring access to a large number of non-adjacent records, there are certain common applications where any other organization would involve a large number of passes of the files and a very complex suite of programs.

4 Indexed sequential: Here the information is held sequentially, having in addition an index which specifies the highest key on each track or block. This enables the required track for any record to be located without the disadvantage of large indexes and is suitable for serial as well as random processing since the records can be accessed in serial order more efficiently than with indexed random accessing.

Indexed sequential has the following advantages over strict serial accessing:

(a) Inactive records need not be handled

(b) With a slight modification to the indexing technique, insertions can be held in an overflow area assigned to each cylinder. This avoids the need to copy a file for each update. Periodic re-organization runs can then be used to restore the file to strict serial sequence and for taking copies for reconstruction purposes

Note: Standard indexed sequential accessing does not obviate the need for complete planning for data security.

5 Address generation: In this case a mathematical manipulation (address algorithm) is performed on the record key to give the disc address of the corresponding record. In the simplest case, there is a direct one for one relationship. Under normal circumstances, it often proves impossible to choose an algorithm for the given key which generates a unique address within the range required. In this event each addressed location (often called a bucket) is made capable of holding two or more records. A practical system usually has to compromise between leaving sufficient space in each bucket to accommodate the maximum possible number of records assigned to it, thereby wasting space, or choosing a more economical packing density, and making provision for records to overflow from their assigned bucket.

Serial and random processing

Whether serial or random processing is used will depend on the type of application.

Three types of processing are considered here.

1 REAL-TIME RANDOM PROCESSING

Here each item of data is input as it becomes available and may have to be dealt with immediately. This automatically implies random processing. Many enquiry systems are of this type and usually external factors decide if this type of processing must be used.

The other types of processing can be called 'batch processing' since the data is submitted in batches for processing at one time.

2 BATCHED RANDOM PROCESSING

Here the data is submitted in batches, each batch containing data in random sequence, and processed by random processing of the file.

3 BATCHED SERIAL PROCESSING

Here the data is submitted in batches, which is either pre-sorted or sorted after initial input to the computer and the main file is then processed serially. This is the type of processing always used on magnetic tape systems.

When the choice is between batched random and batched serial processing then the latter method will often be preferable, due to the large amount of head movement time involved in random processing. This factor does not arise in the case of the magnetic drum which has no head movement, and its effect might be small on any system with several files on the same device, as there will be head movements between the files anyway. If the case is simple though, serial processing is usually faster than random processing unless the activity of the file is less than about 5%. The advantage of serial processing may however be off-set against the need to sort input data.

If the input data is to be matched against more than one main file, held in different orders, the data may be sorted to the order of the largest file and the other file(s) accessed randomly. However, it might be faster to dispense with the data sort and to access all files randomly.

On a job which is basically serial but includes some high priority data, such as enquiries, it may be possible to combine both types of processing. In particular, if the enquiries can be arranged to be submitted with the batched data, the data vet program may be able to isolate them and process them immediately using a random approach. Similarly, if data is being submitted over a transmission network (on-line) a program may read and vet data, processing urgent data immediately and batching the remainder ready for sorting and later serial processing.

Summary

The main points to be considered for structuring a file are:

The file should be designed in general terms, making any necessary assumptions.

This design should be adapted in the light of known factors which affect the file.

A file structured in levels will often have at least three levels, these being an overall file level, a main group level within the file and a sub-group level within each main group, the data items occurring within each sub-group.

Control and reconciliation records can normally be situated at the start and end of levels.

The sequence of records, if not previously defined, should be decided.

KEY SEQUENCE

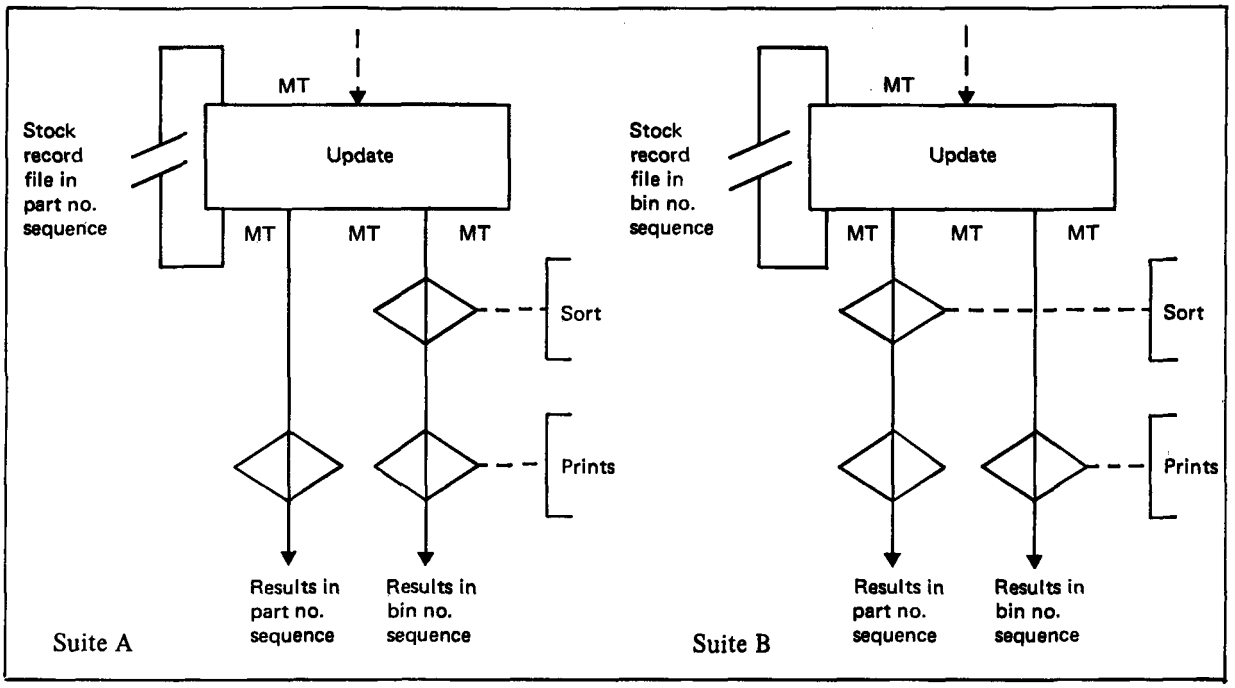
A difficulty which the planner may experience when trying to design files is that of deciding what information is to be used as a key. The key is required to determine the sequence of the records, enabling them to be re-arranged by a sort program from their present sequence to that required by a subsequent process.

The choice of the fields to be used to constitute the key is usually determined by the order in which the results are required, normally, though not always, specified by the systems analyst. Where all types of output results are required in the same basic sequence the records in the master file would be held in this sequence to obviate the necessity for a sort program to precede the print program(s).

In the payroll example, the records in the master file would be in the sequence employee within paying-point within factory if the payslips are to be produced in that order and they are the only output required. If the records are held in any other sequence a sort program would be necessary between the update and print programs to produce the payslips in the required sequence. The decision is not always obvious. Further output may be required in a different sequence. Timings would then have to be performed to determine which sequence of records in the master file would be most economical.

Example of sorting by keys

An early version of a stock control job deals with spare parts stored in bins in a warehouse. Each part has a bin number as well as a part number, and the two numbers bear no relationship to each other. Records in one of the main results files are produced in part number sequence and records in the other main results file are required in bin number sequence. The program suite chart shows two possible arrangements, one for each sequence:



Thus the choice of key is between part number and bin number, and it eventually depends upon the running time of the two sort programs. On this criterion suite A is a preferable choice, the dominant factor being the length of the key used for sorting.

When faced with a situation like this, in which results are required in two or more different sequences, the planner should tell the systems team what savings can be made by modifying the requirements to allow all printed results to be produced in the same sequence. In the stock control example, if a sequence could be found which was satisfactory for both sets of results, a sort program would be unnecessary, giving a considerable saving in running time, particularly if the job is run frequently.

Contents of the key

The key in any record should consist of items which must be present in the record for other reasons. In the stock control example, the part numbers must be present, even if they are not part of the key, for printing on invoices. In a payroll job a department number and an employee number would almost certainly be present in the key and these would also appear on the printed output.

A field of the sort key may contain alphabetic or numeric characters in different records. Great care should be taken to ensure that the sequence required by the system (alphabetic characters taking precedence over numeric characters or vice-versa) is compatible with the collating sequence of the computer.

Comparison of collating sequences (graphics)

1		2		3		4	
System 4	1900	System 4	1900	System 4	1900	System 4	1900
0		Space	:	$	#	—	'
1		£	;	*	£	>	–
2		.	<	)	%	?	.
3		<	=	;	&	:	/
4		(	>	¬	'	#	@
5		+	?	-	(	@	
6		|	Space	/	)	'	
7		&	!		*	=	
8		!	"	,	+	"	
9				%			

5		6		7		8	
System 4	1900	System 4	1900	System 4	1900	System 4	1900
A	A	J	J	S	S	0	[
B	B	K	K	T	T	1	$
C	C	L	L	U	U	2	]
D	D	M	M	V	V	3	↑
E	E	N	N	W	W	4	←
F	F	O	O	X	X	5	
G	G	P	P	Y	Y	6	
H	H	Q	Q	Z	Z	7	
I	I	R	R			8	
						9	

These tables represent the sequence on both ranges, not code equivalents

Summarized the sequence is:

System 4: Space; punctuation and symbols; alphabet; numbers

1900 : Numbers; punctuation, space and symbols; alphabet; more symbols

RECORD DESIGN

The planner must decide the format and sequence of data items within each record. This decision is very important, because record layout and block size are major factors in the running time of the job, and small variations can often have a critical effect upon the running time.

Before starting this stage of file design, the planner must be aware of the possible variations and of their consequences. The various formats for holding each data item must be considered and the ways in which each field can be held within the record. The file activity may also influence the choice for the size of each record.

Data format

Each item of data within a record can be held in a variety of formats. The formats normally available are words, characters (1900 Series), bytes (System 4) and bits. Each item of data should be considered individually and, by careful evaluation, the most suitable method of holding this item in the record will be found. A large amount of space can be saved, for instance, by using a bit to indicate two-way conditions. A 1900 word can accommodate 24 of these conditions and a System 4 word can accommodate 32, one in each bit of the word.

Once each item within the record has been considered the record should be looked at again in its entirety. It may well be that what is best for a particular item is not best for the record as a whole. Re-arrangement of the order of the items within the record may also produce a more beneficial result.

File activity

The team activity used in connection with a file refers to the proportion of its records which are processed in any one run of a program.

Any time spent handling inactive records is wasted, since these records do not contribute to the results. It would appear at first sight impossible to avoid at least reading, examining and writing forward every record, because a record is not found to be inactive for a particular run until it has been read and examined. However, in some jobs there are opportunities of substantially reducing the amount of wasted time.

When it is known that the inactive as well as the active records will have to be read, examined and written, it becomes important to assess the activity of the file carefully, because it exerts a strong influence on the choice of record and block layout. A small error in the assessment may lead to a considerable increase in running time. As far as the inactive records are concerned, nearly all handling time consists of reading and writing; examining the key takes a negligible time. The handling time is minimized by keeping the records as small as possible. However,

it follows that all active records of the same type as the inactive records are also as small as possible and this arrangement almost certainly results in a higher processing time for the active records, due to expanding and contracting the information.

If enough records in the file are active, the extra processing time so incurred will exceed the time saved in input and output.

The difficulty of working out the ideal record layout for a file of given activity is increased if multi-record blocks are used. The distribution of active records in the blocks will almost certainly not be even.

Data arrangement within a record

Data items within a record can be arranged in the following ways:

1 Variable field packing

2 Fixed field packing

3 Integral word packing

4 Combined packing

VARIABLE FIELD PACKING

Each data item occupies the minimum number of character positions which its value requires, non-significant zeros and spaces being suppressed. Items are separated by a specified separator character.

Records in which the items are packed in this way occupy the least possible space (both on the tape and in the input/output buffer). The time required to read or write the records is thus minimized. However, should it be necessary to process individual items in such records in any way, they must be expanded after the record has been read, and packed before it is written. Both the expanding and condensing routines require space in the store to hold tables, but a disadvantage which may be more serious is that they are both comparatively slow. Thus the read/write time is low, but the processing time is high.

A group of items which is usually held in variable field form is a name and address or similar alphabetic fields. Individual items in a name and address are rarely handled independently. Thus expanding and packing time is kept to a minimum.

FIXED FIELD PACKING

Each data item occupies a fixed number of characters, which may be spread over two or more words, regardless of its value. Any redundant character positions are filled with zeros or spaces.

Fixed field data items occupy more space on tape and in the input/output buffer than would the same data item held in variable field form. Thus the read/write time is greater than for variable field data.

INTEGRAL WORD PACKING

Each word may contain one or more items, but no item may be spread over more than one word, unless it is more than one word in length. The greatest benefit is obtained from integral word packing if a complete word is allocated to each item which is frequently used in processing (or two complete words for each such item which is more than one word in length).

Integral word records occupy more space on tape and in the input/output buffer, and the read/write time is appreciably higher than for packing records.

COMBINED PACKING

A combination of the above methods may be held within the same record. It is often desirable to arrange some data items in a record in one way, and some in another. Almost every record contains at least one item which has to be handled by the program (to discover whether the record is active) and which is handled independently of other items.

Sometimes there are other items occurring in each record which are frequently used and which are handled independently of other items. Such items should not be packed without regard to the layout of other items in the record. This ensures that the time taken for frequent processing is kept to a minimum.

It is usually best for integral word data to come first in a record, with fixed field data next and variable field data last. With this arrangement it is unnecessary for time-consuming manipulation to be carried out in order to process integral word and fixed field data. In some languages (System 4 CLEO, for instance) this sequence is compulsory.

INTERNAL HANDLING FORMATS

To consider the effect of the various methods of holding data on the processing time the manner in which data is handled in the computer must be considered.

Computer	*Internal data handling format*	
	Fixed field (almost any length)	*Integral word (fixed) length)*
1900		All types
System 4-30	All types	
Other System 4	Alphanumeric data Decimal data Binary patterns	Binary values Floating point Data

Thus it may be seen that on all machines variable field packing necessitates unpacking and packing of the data on input and output. This not only consumes processing time but also requires core space for the relevant tables. On 1900 Series computers fixed field data also requires unpacking although on System 4 computers it is handled directly.

Sub-record

The record structure already described lends itself very well to the concept of holding the items of data within the record in the form of sub-records. It is a good idea if the data structure of a file reflects the problem structure of the program. Thus, the data in sub-records should reflect the requirements of individual processing tasks.

An additional advantage of this system becomes evident when amendments are required. If a record is structured into sub-records, any changes to the file can be restricted to insertion of new sub-records or amendments of one existing sub-record. In the former case no change will be necessary to existing routines and in the latter, only changes to routines handling the modified sub-record are necessary.

Summary

In designing records one must balance the saving of media space of the various methods against the processing and core requirements, bearing in mind the processing capabilities of the computer.

In order to do this one must consider the manner of holding each data item and the order of the data items within the record. The activity of a file can be a major factor in determining the size of a record. Data records may be held on the file in a number of ways. Each should be carefully evaluated as to its advantages and disadvantages. It is possible that for some jobs a combination of methods will prove to be the most successful in minimizing the running time.

Before the combination is finally decided upon, it must be verified that the proposed arrangement does not contravene any restrictions imposed by the programming language; conversely a programming language may be chosen to match the file handling method required. Structuring each record into sub-records which reflect the problems of the particular program will facilitate any extensions or amendments to that file, as well as aiding the program structure in general.

BLOCK FORMAT

The next stage in the design of a file is to determine the block layout and the block size. This in turn will determine the length of the file and consequently the number of reels or disc cartridges required. The block may contain a single record, two or more records, or a part of a single record and the records maybe of fixed or variable length. The size of the block will also enable the time taken to read and write the file to be assessed, and it will then be possible to estimate the running time of the program.

Record arrangement within a block

There are four possible relationships between records and blocks:

1 Each block contains a single record

2 Each record extends over two or more blocks

3 Each block contains two or more records

4 Records are held in blocks, but without regard to the block length

SINGLE RECORD BLOCKS

This arrangement is used mainly in programs which are calculation dominated. In general, reading and writing time is higher for single record blocks than for multi-record blocks, but this will make no difference to the running time of a program which requires more time for its processing routines than is available from input and output for concurrent running.

Single record blocks may also be used when the input/output buffer cannot be made large enough to accommodate a block containing two records. However, in some cases it is likely that this problem can be solved by spreading the record over more than one block.

MULTI-BLOCK RECORDS

Records are spread over more than one block only when some of the records in the file cannot be wholly accommodated in the input/output buffer area. If records are held on more than one block, the planner should be particularly careful about the order of information within the record. He should avoid such mistakes as holding opening balances in one block and closing balances in a later one, because store space will be wasted in the preservation of the opening balances until the closing balances have been read. Information to be processed first should be part of the first block and so on throughout the record. This technique assumes that the earlier block(s) may be updated without reference to the later block(s).

If some of the information in the record is used very rarely, this should in general be held in the last block or blocks of the record, while an earlier record contains information which would indicate whether, in a particular run, subsequent blocks contain active information or can be read and immediately written or just read and ignored.

For variable field records some indication should be held within each block (other than the last) of a particular record to show that the same record is continued in the following block. If the record is entirely fixed field it is of course known in advance how many blocks constitute a single record, and what information each block contains.

MULTI-RECORD BLOCKS

A block containg two or more complete records is the most common arrangement of blocks and records. In general, blocks should be as large as space available for the input/output buffer allows. This minimizes reading and writing time.

OVERSPILL BLOCKS

Deciding the relationship between records and blocks becomes more difficult when the records may vary in length, especially if the variation is great. For example, assume that the maximum record size is 600 words but 80% of the records are 360 words or less. The input/output area is chosen to be 360 words in length. Each block includes a character which indicates the presence or absence of a continuation block following it. Each continuation block is sufficiently large to hold the remainder of the record. To judge the effectiveness of this system suppose that the average size of those records less than 361 words in length is 300 words, and the average size of the larger records is 420 words. Then, for 80% of the records the input/output buffer is filled on average to $\frac{5}{6}$ capacity, and for the remaining 20% of the records the input/output buffer is used twice, once to full capacity and once on average to $\frac{1}{6}$ capacity.

Thus the effective usage is:

$$\left(\frac{80}{120} \cdot \frac{5}{6} + \frac{20}{120} + \frac{20}{120} \cdot \frac{1}{6}\right) \cdot 100\%$$

which gives 75% usage of the input/output buffer.

If the alternative method of making the input/output buffer 600 words in length is used, then for 80% of the records it is filled on average to ½ capacity, and for 20% of the records it is filled on average to $\frac{7}{10}$ capacity.

In this case the effective usage is:

$$\left(\frac{80}{120} \cdot \frac{1}{2} + \frac{20}{120} \cdot \frac{7}{10}\right) \cdot 100\%$$

which gives 54% usage of the input/output buffer.

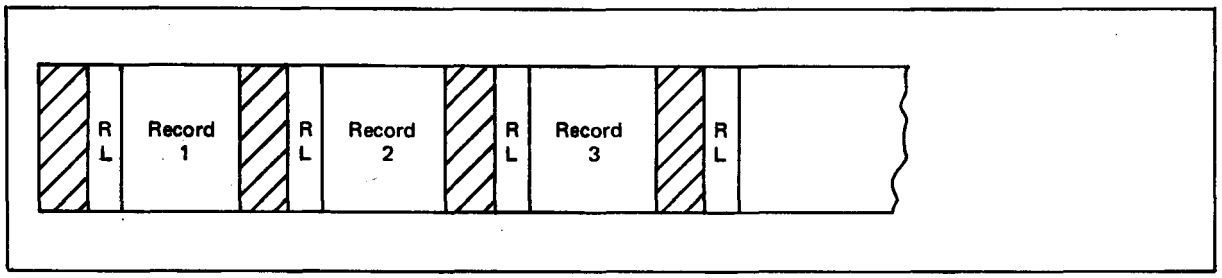
Block layout

The various methods in which magnetic tape records may be composed into blocks are best represented diagrammatically. The methods for 1900 Series and System 4 computers are treated separately, as there are small differences both in layout and terminology.

1900 SERIES

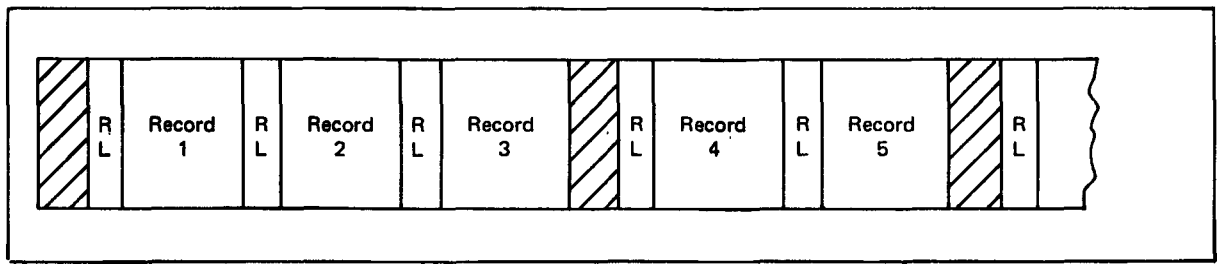
There are two methods available.

1 Unbatched records



Each block contains a single record, and each record has a record length (RL) in the least significant 15 bits of the first word of the record. This length includes the first word.

2 Batched records

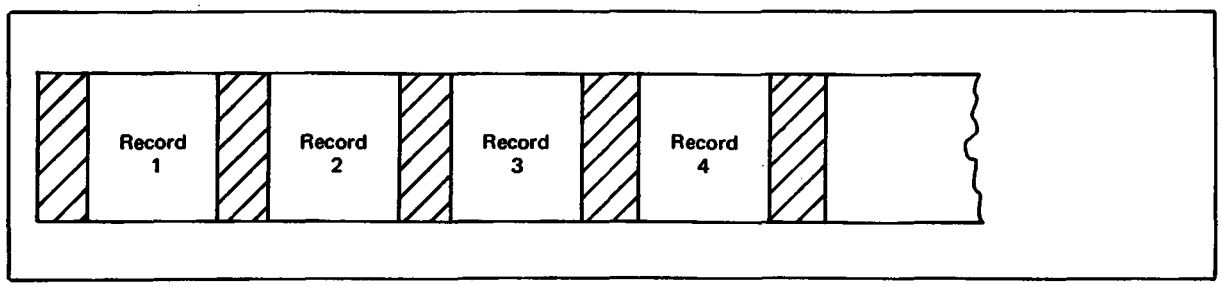


Each block may contain one or more records, and each record has a record length field as for unbatched records. There is no block length field at the beginning of the block.

SYSTEM 4

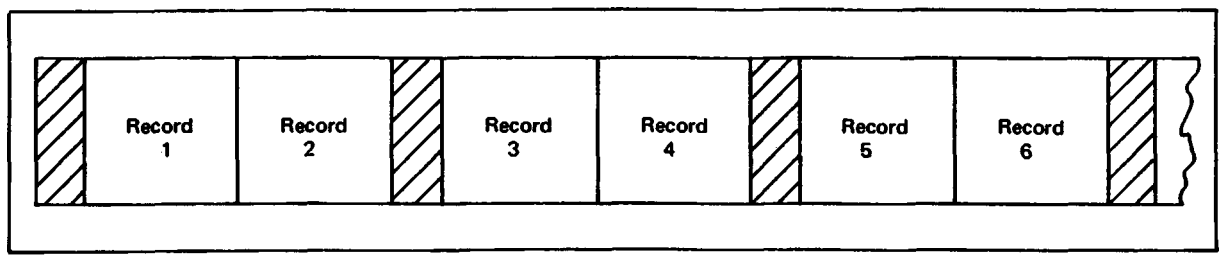
There are four methods available.

1 Fixed length unblocked records



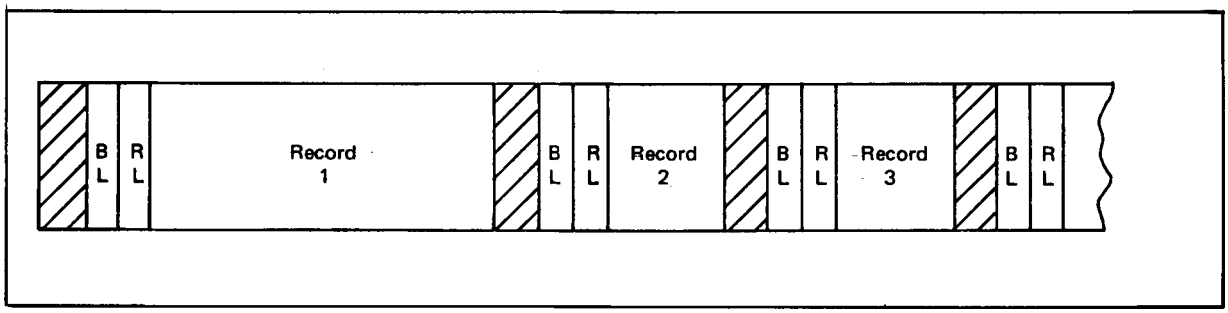
All the records are the same length and each block consists of a single record.

2 Fixed length blocked records



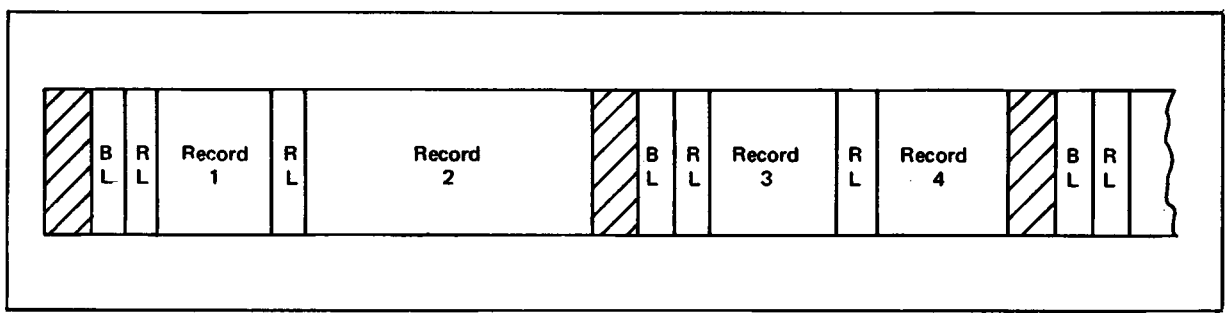
All the records are the same length but each block may contain several records.

3 Variable length unblocked records



Each block contains a single record, but records are not all the same length. The first four bytes of each block comprise the block length field (BL), the first four bytes of each record comprise the record length field (RL). Each of these four byte fields contains the appropriate value, in binary, in the first two bytes and space characters in the remaining two bytes. The record length value includes the four bytes of the record length field, and the block length value includes the four bytes of the block length field.

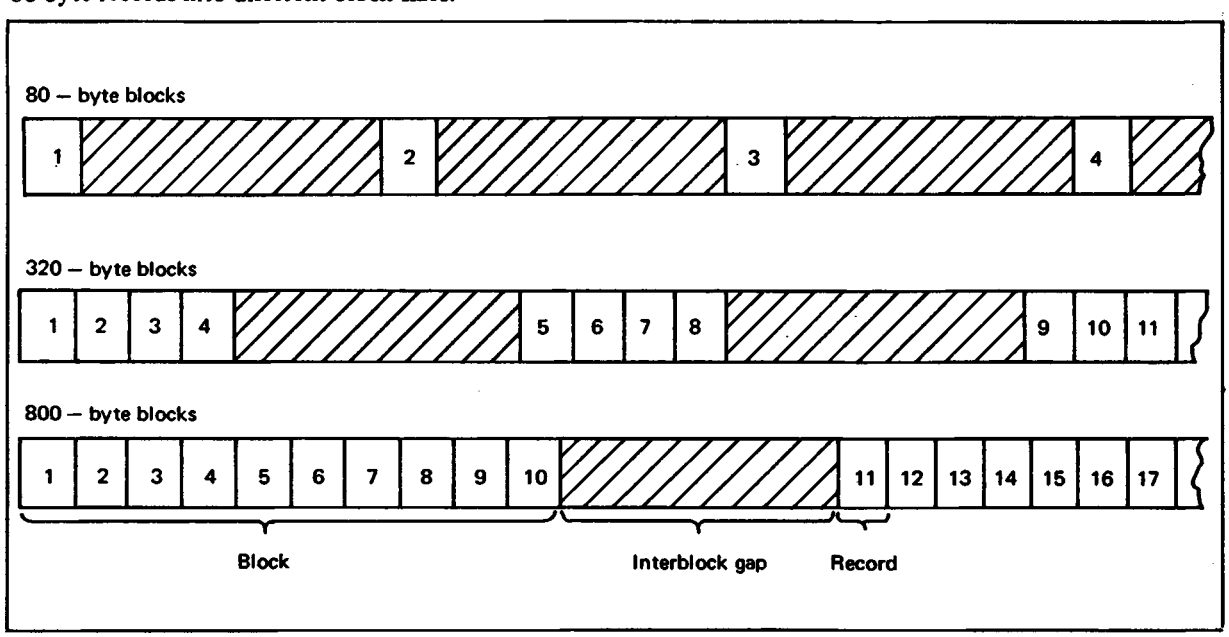
4 Variable length blocked records



Each block may contain one or more records, and the records are not all of the same length. Each block commences with a block length field and each record with a record length field, as for variable length unblocked records.

FILE LENGTH

When deciding upon the way in which records and blocks will be related on a file, the planner will automatically determine the length of the file. If the block size is small, the additional number of blocks and interblock gaps increases the length of the file. The following example illustrates the effects on the physical file length of packing 80 byte records into different block sizes.



If a file occupies more than one reel of magnetic tape the additional overhead of time taken to rewind and change the reel will be added to the file passing time. This overhead can be avoided providing there are sufficient tape decks available by preloading the files, that is, loading a reel on another deck before the previous reel has been completely read.

If the planner has the choice of two file layouts, one of which ensures that one reel will suffice to hold the complete file and the other will not, the former will normally be chosen, but it must be decided whether the time saved by avoiding a reel change is greater than that wasted by using a layout which is inconvenient in other respects. With larger files spread over many reels, preloading will almost certainly be specified if two tape decks are available and the point mentioned above will not arise. However, since tape decks rewind at twice the maximum reading speed, it may be more economical to rewind and reload, if the additional tape deck could profitably be used for another program.

The planner should remember that one or two superfluous characters in a record will add considerably to the file length if the file contains several thousand or million records.

The planner must also remember that the file length which he has to consider is the length when the system is fully implemented and that a file which is well under one reel in length during the first few months of computer processing may well grow into several reels at a later date. The growth factor should be considered.

Timing

It is necessary to estimate reading and writing times and the time of any processing that may occur so that the maximum advantage of concurrent working can be taken. Speeds of input and output equipment are generally available in the appropriate hardware peripherals manual, and the time taken by individual processing actions can also be found.

A warning must be given about input and output timing. The reading and writing speeds usually quoted are the theoretical maximum speed and an average effective speed. They are not intended for use when accurate timings for a particular file are required. Instead, the planner should use a formula to give the effective transfer rate, relating the reading and writing speed to the block size. The effective data transfer rate is a function of the nominal data transfer rate (or reading/writing speed), the length of the interblock gap, and the block size, and may be calculated using the following formula:

$$\text{Effective data transfer rate} = \frac{n}{i+n} \times m \text{ characters or bytes/sec}$$

where n is the number of characters or bytes in a block.

i is the length of the interblock gap in characters or bytes.

m is the nominal data transfer rate in characters or bytes per second.

However, even the speed given by this formula must be modified in some cases. The device being timed may not always gain immediate access to the channel if other devices are using it. If the other devices have comparatively few blocks, then timing of the device under consideration will not be much affected, but if they have a large number of blocks, and write actions on the various tapes in the system are not evenly spaced, then the timing will be considerably affected. The latter case can influence the design of the file, because in general it allows processing time to be higher and yet still be concurrent with input and output. Thus it may be possible to condense information on the file which would otherwise have been held in fixed field or integral word format because there was not thought to be sufficient processing time available.

ACCESS TECHNIQUES

Accessing inactive records on serial files

In some files nearly all the time spent in processing is on inactive records. This is wasted time because it does not contribute to the production of results, so it becomes particularly advantageous to reduce this time wherever possible. Some methods for achieving this are now described.

INSPECTING GROUPED RECORDS

For files which are not highly active the time spent searching for active records can be reduced considerably by using the following technique, which applies to files in which there are two or more records in each blocks. The method consists of holding at the beginning of each block the record key with the highest value in that block.

After the block is read, the key is examined. If it is less than the key of the next amendment, then none of the records in the block can be active, and the block is written forward.

This technique can influence the planner's decision about block length. If records are short and there is plenty of store space available, it may be possible to have blocks of thirty or forty records each. However such blocks cannot be dealt with effectively by the technique described, since even in very low activity files there will be few blocks without at least one active record. The average number of records in each block should be such that a worthwhile proportion of blocks can be expected to contain only inactive records, provided that this length of block does not result in an unsatisfactory balance between reading and processing time.

CHANGE FILE TECHNIQUE

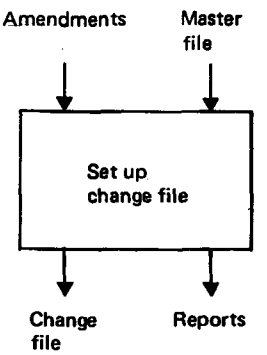
In most jobs the master file is updated only on occasions when information is required for output. However, very large files often have to be updated two or three times before the file is used to provide results.

Example of change file technique

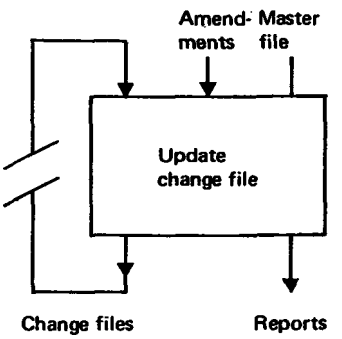
A file contains 1,000,000 records, which is used to provide results every four weeks. The average number of amendments that arises in each four-week period is 200,000. This represents a very large volume of data vetting, sorting and updating which will almost certainly be spread over the whole period.

If processing is done daily, the average number of records to be processed each day is about 7,000. If the file were updated every day, the program would have to write forward 1,000,000 records in order to amend 7,000 records. The change file system overcomes the problem by three stages of processing.

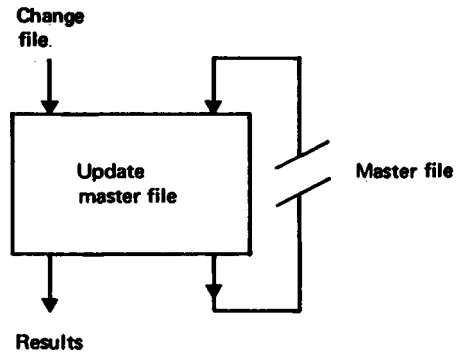
1 Each of the first day's amendments (previously vetted and sorted) is read and the master file searched for the record to be amended. The amended version of the record is written on the change file.



2 On each subsequent day, each amendment is read, and applied to the appropriate record. If this record has previously been amended, it will be on the brought forward version of the change file; otherwise it will be on the master file. The newly amended record is written on to the carry forward change file.



3 After the change file has been updated on the 28th day, it is used to amend the master file.



In the example it is apparent that the change file system offers great economy of running time. In jobs where the number of records on the master file is not so large, nor the number of amendments so comparatively small, the planner must estimate the running time of both a change file system and a normal updating system before deciding which to use for the job under consideration.

DIVISION OF FILES

Another solution to the problem of inactive records is to submit them for processing less frequently than the active records. This can only be done if it is known in advance which records are and which are not likely to be active frequently.

Example of division of files

A job is concerned with preparing invoices for a manufacturer's customers. The number of customers is very large, but only 10% are considered to be major customers, that is customers to whom deliveries are made, and hence invoices sent, frequently. Invoices are prepared by the computer, and the master file contains one record for each customer. The file is processed daily, but on average is only 15% active. Therefore time is wasted in copying the other 85% of the file on each run. However, if invoices for major customers are processed daily, the remainder only need be processed weekly. The only preparation required on the computer is to divide the file into two, one containing records for major customers, the other for minor customers, and a great saving in running time is obtained.

Modes of access

Magnetic tape is essentially a serial medium for holding data, and access must also be of a serial nature. Of much more importance is the method of accessing data on direct access files. On direct access media files may be processed serially, sequentially via indexes or pointers, or accessed directly via indexes or address generation techniques.

FILE CONTROL

Although computer equipment is increasing in reliability, thus reducing the chances of machine errors, human errors will always occur, and therefore file control procedures are a necessity for all installations.

The main reason for control is to ensure that correct data files are loaded. It is always possible for an operator to make an error in file loading, particularly in a multiprogramming environment. It is therefore necessary for each file to have a unique identification which can be checked by the program at run time. In addition, for input tape files a check on the reel number is essential so that the correct loading sequence is maintained. For output magnetic tape files there must be security checks to ensure that data is not overwritten before it is obsolete. These types of control are required in all installations and are not connected with the efficiency of the equipment.

Error checks are necessary in order to ensure that incorrect data, caused either by a malfunction of the equipment or, as is more likely, by a fault in the recording medium, cannot be accepted without warning of the error condition. Controls and error checking procedures are provided by software routines so that they do not have to be rethought for each program, thus avoiding wastage of programming effort, ensuring more efficient procedures and standardizing operating procedures. Obviously the software will require standard identification

for files and will also allow standard error checks and recoveries. More flexibility can be obtained if required options are provided for users' programming routines.

These software checks are for operator and machine errors and do not replace program reconciliations which are required to ensure that all data has been processed correctly and consistently throughout a suite of programs.

Protection for direct access files

Two facets of disc handling make the approach to file control on disc different from magnetic tape. Firstly, while a tape will normally hold one file or part of a file, it will not be unusual for a single disc pack to hold several files concurrently. Therefore while with magnetic tape access to a file can be restricted by off-line control of the spool(s) of tape on which it resides, for a file on disc the problems of off-line control are magnified.

The second point is that on tape, once a file is set up it remains unchanged until the file is released and the tape re-used. Standard labels written to the tape when it was initially set up uniquely define the set of data on the spool. On discs, as on magnetic tape, a file is identified by its file identifier. When the file is updated in situ, the file at the end of the updating run differs from the file at the start of the run, but the file identification remains the same. Thus the identification no longer defines the data set uniquely. Standard label checking has no method of distinguishing between a newly-created file and the same file after it has been updated several times.

Furthermore when the file is created an expiry date is specified. Before this date is reached the file is totally protected, that is, it is a read-only file. After this date is reached it may be updated but it is also totally un-protected and may be overwritten at any time.

The problem of random access file protection falls into two areas:

File control

File protection

FILE CONTROL

The most obvious approach to implementing both a file control and a file protection system is to make use of the user label or area facility. This does not exist on System 4 indexed sequential files, but this approach can be modified slightly. Standard file control marks each file with the run number of the program which created it, and also the program name forms part of the file identifier. The additional information needed to implement full file control is:

1 The number of times the file has been updated

2 The run-number of the last program reading or updating it

3 The name of the last program reading or updating it

4 The sequence of programs which have accessed it

The control system implemented may not need all the information but some will certainly be required. In this way any program accessing the file may ensure that the file has been updated the correct number of times, or that it has already been processed by the correct program.

FILE PROTECTION

There must be control to ensure that a file is written to only by certain programs and that a file is destroyed only after a specified date.

The user labels may contain an indication of which programs may open a file for updating, so that the label checking routine may reject the file if the program is not permitted to alter the file (or not permitted to alter the file at this date).

On System 4 computers a file may be deleted in one of two ways after its expiry date has passed: by closing it in a program with the keyword RELSE, or by submitting a DELETE parameter to disc housekeeping (which achieves the deletion in the same way). A simple extension to the method described for file control will prevent any program from deleting any file which it is not entitled to, but disc housekeeping may be used to delete any file whose data has expired. In order to avoid misuse of this latter facility, a file deletion program may be designed and used to process all file deletions. This would read parameters to find which files may be deleted, open them, check that they are ready for deletion and close them with the keyword RELSE.

INDEXED SEQUENTIAL FILES ON SYSTEM 4

The foregoing suggestions would be implemented using the user area or label facility. However, this is not available on indexed sequential files. The same system could effectively be implemented on these files by holding the corresponding data in the first and last blocks on the file, with dummy keys.

Example

$(00)_{16}$ ----------$(00)_{16}(01)_{16}$ and $(FF)_{16}$ ------$(FF)_{16}(FE)_{16}$

IMPLEMENTING FILE CONTROL AND PROTECTION

Any such system must derive its strength from the fact that it is standard, and its rules apply equally to all files and programs. The system chosen should be implemented in all programs for an individual installation. This procedure which has three advantages:

1 Increased protection

2 Standard operating procedures

3 Coding need only be implemented once

This last point may be achieved either by writing the control routine entered on opening and closing files as a macro or by compiling a standard check routine as an independent module or segment which may be linked to all disc handling programs.

RECONCILIATIONS

The purpose of reconciliations is to ensure that data which has been accepted by the computer system is not lost, duplicated or corrupted. This could easily be done in a system where the final output should correspond completely to the initial input; unfortunately, most computer projects are more sophisticated than this and reconciliations have to be designed individually for each file or program.

By far the most frequent cause of reconciliation failure is a program error. Reconciliations should be designed to detect an error at the earliest point possible, with the aim of minimizing reprocessing.

There are three main types of reconciliation:

1 Validation of the contents of a file held on backing store

2 Validation of an updating process

3 Validation of a calculation process

The first type of reconciliation consists of accumulating control totals as the file is written, recording these totals at the end of the file, accumulating the totals again when the file is read (either by a different program or, in the case of a brought forward/carried forward file, by the following run of the same program) and comparing the totals held on the file with the newly accumulated totals.

The second type of reconciliation consists of accumulating totals for applied amendments and records written to the carried forward file and relating these to the totals held on the brought forward file by checking that the brought forward total + amendments total = carried forward total.

The third type of reconciliation is not normally required in commerical programs. It may sometimes be desirable when statistics are being accumulated to various positions of a stored matrix to incorporate a check, which might be the sum of columns (or rows) = control total of input data. This can be considered as an extreme case of the second type of reconciliation.

Another use occurs when rounding off of individual items is being performed after a calculation and where a comparison may be made with the result of rounding after the same calculation is performed on the sum of the items. This is mainly useful where the calculation is fairly involved.

It is important to understand what these reconciliations achieve. The first proves that all records which were thought to have been written to the file were in fact written. The second that all records which were written have been recognized by the reading program. The second proves in addition that all inactive brought forward records have been copied to the carried forward file and that changes to the main file are consistent with the accepted current data. The third reconciliation shows that important quantities or values have not been corrupted. In each

case the reconciliations prove that the contents of the files are consistent with the processing that has been carried out, not that this processing was correct.

Both of the first two types of reconciliation should always be incorporated in commercial programs. However, there is some choice as to the number of control totals for each file.

A count of the number of blocks on the file will often be maintained by software routines. In some cases it may be required to keep a count of records on the file. Beyond this totals may be accumulated for important values and quantities within each record.

A balance must be struck between the increased possibility of detecting processing errors and the cost of providing additional reconciliations; maximum use should be made of totals which have to be accumulated for other purposes. To avoid the possibility of compensating errors, cases where amounts are added to and subtracted from the same total should be minimized, separate counts being kept of positive and negative amounts wherever possible.

Reconciliation requirements should be written into the program specifications with particular care, as this is a field where ambiguity flourishes, with dire consequences. The action to be taken in the case of a reconciliation failure should be written into the operating instructions; it is very seldom that the operator should be given the option of continuing, since a program which has failed to reconcile correctly is generally suspect. The fault should be ascertained and cleared before the program is used again.

When preparing estimates for the running time of a program or suite, the time taken to accumulate control totals should be assessed and included in the total.

Reconciliations with restarts and re-runs

With files which have been set up so that they can accommodate restarts and re-runs, reconciliations can also be performed for each group of data. There may be a natural grouping in the data. The checking by restart group can then consist of checking for each natural group that falls entirely within the restart group and checking the totals to date for any natural group which overspills the end of a restart group. There must then be a check that all natural groups of data have been processed. Overall totals may be in the process of being formed for statistical or outside control purposes. By making all of these the totals which check the validity of re-runs, the validity of already despatched totals will be ensured if it becomes necessary to perform a re-run.

The standard principle of checking file updating is by means of a reconciliation of the form:

Brought forward file control total + current file control total ± amendments not arising directly from the current file = the sum of items written to the carried forward file.

With the reel changes of the main output file as the basis of restarts and re-runs there is a difficulty in that the input control figures are not available for the restart groups. It may be necessary to divide the check into:

1 Brought forward + current ± amendments = carry forward

   where all figures are accumulated over the restart group.

2 Brought forward control = brought forward accumulated

3 Current control = current accumulated

   where in each case a grouping on the input file is the basis of the points at which the check is made.

   In case 3 the group may be the complete file.

This is justified on the grounds that the normal reason for restarting or re-running is not the failure of reconciliations and hence the detection of such a failure is not logically tied to the restart group.

It should nevertheless be possible to check the validity of reconciliations (2) and (3) on a re-run. This will mean holding totals for the input file groupings to date, since some totals will be checked on a re-run. It is extremely important that all totals are fully defined as to their contents and layout. It must, for example, be clear what is being added into the total for each item and over what data the total is accumulated.

The reconciliation problem is present to a lesser extent in file search programs with respect to the subsidiary input file(s) and may be solved in a similar way.

SUMMARY

The planner's main aim when designing files should be to allow each file to be processed as quickly as possible without demanding too much store space for the input/output areas or the file handling routines.

The best approach is to determine a structure which gives either the minimum reading or writing time, or the minimum processing time.

The minimum reading or writing time should be used as a basis, when:

1 The file is to be submitted to a sort program or some other program where read/write time must be minimized.

2 The activity of the file is calculated to be less than about 10%. Blocks, and therefore buffer areas, should be made as large as possible. Records should be kept as small as possible.

The minimum possible processing time should be used as the basis for all other files. The only items which should be held in condensed form are those which are rarely handled independently. Single record blocks may be used to reduce overall processing time unless their length is likely to vary too much.

When the planner has designed the structure to be used as a basis, the total time needed for processing the file must be calculated. The planner must be prepared to modify the structure until an acceptable time is achieved. Having determined the structure of the file, the planner should provide a high degree of control so that the file cannot be corrupted, and should ensure that suitable facilities are available for reconstructing any damaged files.

Appendix 6 Program specification

A program specification is in three sections. The first two sections are the same for all programs in the suite. They give the overall picture of the project and the information should be extracted from the systems or outline suite specification. The remaining section gives the detailed information for each particular program. Where the inclusion of detail in a section would interrupt the flow of the narrative the information should be included in an appendix which would be referred to in the section itself.

Suite introduction

This section is intended to give the programmers a general idea of the overall scope of the project. The section is in three parts.

BACKGROUND OF THE PROJECT

A general description of the organization or department for which the job is to be done, and in particular of those areas and procedures which the job will affect. Details of the departmental communication system for this project may also be included.

THE JOB TO BE DONE

An explanation of the need for the principal tasks and a brief description of these tasks.

GLOSSARY OF TERMS

Definitions of all terms used in the suite which are not likely to be self-explanatory.

Suite organization

This section is intended to show the programmers the overall design. The section is in two parts.

DIVISION INTO PROGRAMS

A brief description of the purpose of each program and how the programs relate to each other.

ORGANIZATION CHART

A copy of the suite organization chart showing all input and output and when each program is run.

Program description

This section gives the programmer the detailed information which is needed in order to produce the program. The section is in thirteen parts.

IDENTIFICATION

The program name and identity which are to be used in all references to the program.

INTRODUCTION

An explanation of the purpose of the program including the following:

Main results to be provided for external use

Results to be provided for input to other programs

Data provided externally

Data provided by other programs

Times when program is run

CHARACTERISTICS

This includes the programming language to be used and the estimated computer run time. When a program is being divided into routines which are to be treated as independent entities, or where a program is to be segmented, a list of the names of each module or segment and the language to be used should be given.

TASKS

A brief statement of the procedures to be carried out by the program, given approximately in the order that they occur.

DATA

A detailed account of all input files. For each file the following information must be given when applicable:

Medium (cards, magnetic tape, etc.)

File identity

Type of labels, (standard, non-standard)

Block size and record format

Constituent records and sequence on file

Volume

Source

For each record in the file the following information is required:

Number of occurences

Constituent items–name

size (including fixed or variable or column numbers)

units in which held

range, (maximum and minimum values)

optional or mandatory

Specimens of data forms should also be included if available.

The record details and specimens should be relegated to an appendix.

RESULTS

A detailed account of all output files. The information required is the same as in sub-section *Data*, with Source replaced by Destination. The record information for print files is best shown on a print layout chart and must include details of zero suppression required and number of line feeds.

Specimens of pre-printed stationery should be included if available.

PROCESSING

A description of all processing required to convert the data to the required results. Where the program is divided into independent routines the processing should be described for each routine.

The details which should be included are:

Special start procedures

Validity checks for input data

Consistency checks

Formulae for calculations including rounding required

Check digit calculations

Use of tables

File handling, particularly if not using standard software

Pre-printed stationery alignment routines

Complex condition testing, or matching

Main path processing and any optimization requirements

Details of irregular processing runs

Special end procedures

Notes on segmentation

This information should be presented in the most convenient form for the subject, for example notes, formulae, decision tables.

RECONCILIATION AND CHECKS

Details of check totals and methods of reconciliation, for example, if a hash total check is required and exact details of the items to be included must be given, or if an input/output reconciliation is necessary the exact fields to be included and the comparison to be made must be given.

RESTARTS AND RE-RUNS

A description of the restart or re-run procedures which are to be incorporated into the program. Details must be given of the output to be made in order to be able to restart, and also of the method by which the program is to be restarted or re-run. The data grouping for restarts and re-runs must be included.

ERROR CONDITIONS

Details of error conditions which are the communication with the operator. The circumstances giving rise to the condition, the log message, and the alternative actions to be allowed must be included.

OUTLINE FLOWCHART

A copy of the outline chart, showing the basic structure and logic of the program.

TRIALS PLAN

A list of the various cases, and combinations of these, which must be tested, for example insertion, deletion, insertion followed by amendment. Error conditions should also be included. Details of how data is to be grouped for different stages of testing, for example normal, error conditions.

APPENDICES

File and print layouts and any other detailed information.

Supporting documentation which is produced while the program is being developed can be inserted. Forms are obtainable from ICL. Specimen forms are shown on the following four pages.

ICL Data processing **Programming stationery**

Use			
Title		Program/segment ref.	
Prepared by:	Date:	Checked by:	Date:

Page of

FORM C5/25(7.69)

ICL Data processing

Count/modifier record

Programmer | Program/segment ref.

Loc.	Description of use	Content	Set by (F.C. ref.)	Mod. by (F.C. ref.)	Amount	Tested by (F.C. ref.)

FORM C14/14(11.68)

ICL Data processing **Switch record**

Programmer _____

Program/Segment ref.

Switch number	Description of use	Initial cond.	Set by (F.C. ref.)	Unset by (F.C.ref.)	Tested by (F.C. ref.)

FORM C14/13(9.68)

ICL Data processing

Halt and entry point record

Programmer

Program/Segment ref.

F.C. ref.	Action (Specify halt, entry point or displayed message)

FORM C14/15(9.68)

Appendix 7 Routine specification

The contents of a routine specification should in most cases be sufficient for a programmer to write the routine without referring to the program specification. However programmers writing routines should always have access to the program specification so that they can see where their routines fit into the whole program.

The headings and content to be included are as follows:

Identification

The routine name and identity to be used in all references to the routine.

Characteristics

This includes the language to be used and, except for subroutines, the name of the higher level calling routine.

Interface definition

A detailed list of all the higher level external references to be used by this routine. This list should contain the following for each reference:

Reference number

Name to be used. This is normally the external reference but for subroutines may be an internal reference.

Format of item

Contents on entry and exit

Processing

Details of processes to be carried out. This can be extracted from the program specification or can be references to the appropriate sections of the program specification.

Flowchart

A copy of the higher level flowchart containing this routine, or the highest level chart for this routine.

Supporting documentation

This section is built up during the writing of the routine and should include any information which may be of help in maintenance:

A list of called routines and subroutines

A list of data references used in calling lower level routines

Appendix 8 Restarts and re-runs

Throughout this appendix *volume* is used to mean a physically discrete quantity of a particular data storage medium, such as a magnetic tape reel.

This appendix discusses the advantages and disadvantages of using various methods of restarts and re-runs, and suggests techniques which optimize these methods.

The need for restarts and re-runs occurs when it is necessary to repeat the processing of a program run. There are two options: firstly where the original processing has been terminated before completion; secondly where the original run has been completed but it becomes necessary to repeat part of it. A restart will enable processing which has been terminated before completion to recommence from a specified point, other than the beginning, and to continue until the end of data has been reached. A re-run will enable processing which has been completed to recommence from a specified point, including the beginning, and to continue to a further specified point, other than the end of data. If the program is to continue to the end of data, the run should be processed as a restart or a normal run.

RESTART GROUPS AND POINTS

To permit the use of restart and re-run procedures, information about the files, store areas and registers associated with the program is preserved at certain points during the run, to allow them to be set up at some later date exactly as they were at this point. This information may be dumped to a file associated with the program, possibly but not necessarily used only for that purpose. In the case of disc operating systems this file may be held on the system disc.

At the beginning of a restart or re-run this information is used to set up the job in the computer in the same state as it was at that point in the original run: files are aligned, registers restored and so on. Processing then continues normally, ending at the proper run end in the case of a restart and at another intermediate point in the case of a re-run.

The information between two points is known as a restart group, and the points themselves are called restart points.

Types of restart groups

Care must be exercised in dividing the data into suitable groups for restarting and re-running purposes. The method which is selected will depend on the type of program and the amount of data. There are three main ways available to the planner for making the division.

TIME GROUP

In programs which involve a large amount of calculation but have relatively small files the group can be made to correspond to a period of time, for example, the length of time taken to go round the main loop a specified number of times, or the time taken to perform a certain amount of calculating. Before using the clock on the machine as a basis for arriving at the time which has elapsed since the last restart point, the effect of multi-programming should be considered.

LOGICAL FILE GROUP

In many programs the input or output data automatically falls into separate groups of data within the complete file, such as staffing departments within a company or ledger groups. These may provide acceptable restart groups, where each group takes approximately the same time to process and this processing time is suitable for restart and re-run purposes.

PHYSICAL FILE GROUP

A further type of group is the physical file group. In a program with punched card input data for instance, each box of cards can be used to denote a restart group. Similarly for a multireel magnetic tape file each reel of the tape can be used as a restart group. Thus where additional programming for a file might be expected in any case, the physieal divisions of the file can be used as the basis of restart and re-run groups, provided the time taken to process these divisions of the file is found to fall within acceptable limits.

CHOICE OF RESTART GROUP

When considering the placing of restart/re-run groups it is the processing time which largely determines the size of the group. The planner must decide what is to be the optimum time interval for each group.

It is generally accepted that if a program will run for less than half an hour it does not need restart and re-run facilities. The saving in cost of the extra programming and housekeeping involved outweighs the advantages of including such facilities. There is a running time overhead as well when the restart and re-run facilities are included in a program. This overhead will depend partly on the amount of information which is to be dumped. The most acceptable time between restart/re-run groups would depend on the requirements of the installation, but for most installations would be between 10 and 20 minute groups. The running time for restart/re-run groups for data vetting programs is likely to be appreciably less.

It is worth noting that if a program has a low priority when multiprogramming, then groups will probably be processed more slowly than when single programming. In view of this it is possible that restart groups occurring at about 5 minute intervals when single programming would give an acceptable running time per group under multiprogramming conditions.

This time constraint is essential not only for the initial file structure, but it must continue during alterations to the data in the files, including any insertions and deletions. In considering where the restart points are to occur it is also necessary to determine the amount of extra processing that will be implied in the original run at each restart point, at the point of commencing a restart or re-run and at the point of terminating a re-run. A balance has to be struck between the need to minimize normal running time and the need to minimize inconvenience and cost of restarting and re-running.

When the restart structure is based on a file group, it must be decided which file is to be considered.

Input file

The restart group is a subdivision of one of the input files. Programs which are suitable for this method are for example, a program that searches a large input magnetic tape file, extracts a small amount of data, and writes this onto an output file; a data vetting program which reads data from paper tape and writes the vetted data onto a magnetic tape output file. With this type of group the information required to perform a restart of a re-run will be dumped onto one of the output files, with some form of identification so that the point can be located when necessary.

Output file

The group is a subdivision of one of the output files. Programs which are suitable for this method are, for example, a program updating a multireel magnetic tape file where each reel of the updated file could be regarded as a group and as such would be headed by restart information; a file maintenance program where the group is based on the main file being produced. In this type of group the dumped information is usually held at the point where each group begins on the output file.

Economics of restart groups

When considering time intervals, it should be remembered that frequent restart points will increase the amount of storage space required on the dumping medium, and the program will probably become peripheral-dominated. This increase in storage space may be a crucial factor if the file security method used is the Grandfather, Father, Son technique.

Summary

Whichever method of grouping is used, there are certain conditions which should be remembered. The groups should take approximately the same processing time, the physical size of the groups should remain the same in spite of alterations to the file, or if a group size changes, all groups should change proportionately. If the time interval increases beyond the acceptable limit, then an easy method must exist for adjusting the groupings.

In some jobs there will be a fairly obvious natural division in the data which can provide a ready-made restart structure. In other jobs it will be possible to use the physical divisions in one of the data files. Otherwise the planner will have to select groups according to the amount of processing that can be carried out within the time interval allowed for one group.

DUMPING RESTART INFORMATION

There are two approaches to dumping information for restarts. The first is to dump the program so that it may subsequently be re-fed. The second approach is to assume that the program is fed in as for a normal run when a restart run is required. In this case the initializing routines of the restart entry procedures must set up any special conditions.

Dumping the program

The entire program area, registers, file information and so on are output. A restart is initiated by reading the dump back into the same area of store, setting up registers, aligning files, and entering the program at the point specified for restart entry. This is all achieved in PLAN by means of the Dump facility, and in Usercode by means of the Checkpoint facility, in conjunction with the restart facilities. This method of controlling restarts has the advantage that it minimizes the extra programming required in most programs. This is particularly important for complex programs which alter themselves in store. The Dump and Checkpoint facilities are supplied as standard ICL software, so require very little user programming to output the restart information and initiate the restart. Also, by using standard ICL software for handling restarts, operating instructions and procedures are simplified.

A disadvantage of this approach is that much of the information that is dumped is not required for restart purposes, and this takes up a large amount of storage space. This may mean that a separate file has to be used to hold the dumped information. If a magnetic tape file is to be used for this purpose another tape deck would be needed, although if the dump file is to be held on disc it could use a cartridge already associated with the program. If only the most recent dumps are required, as a safeguard during the run, but not to be kept once the run ends satisfactorily, they can, under disc operating systems, be output to the system disc. Normally only the latest dump would be available, because in order to save space, each dump would over-write the previous dump. The program dumping method has the weakness inherent in re-feeding the program from a source where it may have become corrupted and causes an additional overhead at every restart point rather than just at the point of restarting. The Checkpoint facility has the additional disadvantage that if a restart becomes necessary because of a program error, then a restart can only be achieved by amending the program and restarting it at the beginning of the run.

Dumping specific information

The program only outputs information about the registers, files and the current state of running totals, indicators, flags and so on. Thus by outputting only that information which is essential to the setting up of a restart the core store required can be significantly reduced. A restart is performed by loading the program in the normal way, but indicating by means of a parameter that it is not a normal run. The program contains its own file alignment and setup routines, written by the user.

This method of restarting also has its disadvantages, the principal one being the amount of extra programming involved, with consequent increases in implementation time and storage space. The increase in storage space could be overcome to a certain extent by using overlay techniques. Whether or not a separate file is used for restart information again depends upon the space available on the normal output files compared with the availability of magnetic tape decks or disc space as alternatives. This method may also be rejected because of the non-standard aspects of operating procedures which would be required. However if the advantages of such a scheme outweigh the disadvantages then it could be made the standard for a particular installation.

RE-RUN TECHNIQUES

A re-run is made on multi-volume and on lengthy input processes when there is a need to reprocess parts of a file after the initial file has been completed. A re-run may process either a single volume of a multi-volume file, or part of a single volume. In the first case the re-run would replace the old volume with the new. In the second case the re-run would be made onto a new volume.

Merging the re-run results

The ease of merging re-run results depends largely on the nature of the data groups and on the media holding the results. There should be no problems in merging the results from a print program re-run, the only possible errors occurring in the clerical control required to prevent the old results being distributed.

There are two approaches to merging which should be considered:

1 The user program is written in such a way that the results from the re-run and the original results are merged during the re-run of the user program

2 The re-run results are merged by a separate merge program, which is probably part of the installation software

MERGING IN THE USER PROGRAM

Multi-group volumes

If a single volume contains more than one restart group, the original results for each group are copied to the new file until the start of the group to be reprocessed is recognized. The re-run data is then processed and new results produced and output to the new file. The original results for this group are omitted and the results for the remainder of the original groups are copied to the new file, running totals and reconciliation figures being updated. This method could allow any number of non-consecutive groups to be re-run at one time, provided that they are submitted in numerical order.

It might be wasteful, if not impossible, to start this method using restart software, since output files would be aligned to the point of the restart, and would have to be re-aligned and opened for input for copying to take place. The amount of wastage depends to some extent on which system is being used for restarts. If magnetic tapes are used then the time wasted is considerable; with discs only head movement is involved. Against this time wastage should be weighed such points as the time involved in programming start routines and the expected frequency of re-runs.

Another objection to this method is that every file used for output for a normal run must use a new file for a re-run. Although this is feasible, it may be considered operationally undesirable, expecially if there is more than one output file. If the files were held on magnetic tape, then twice the number of tape decks would be required. If they were held on disc, it might be possible to hold the old and new files on the same disc, but the input/output timings would be considerably increased. If the old and new restart groups were the same size, updating in situ might be possible.

On fast processors, the time used to copy is the same as re-running the complete program unless the re-run procedures are designed such that only the volume containing the current restart group is copied.

Single-group volumes

If each volume holds one restart group and begins with the restart information for that group, the merging process required is basically a matter of substituting the new volume for the old. However, in some cases, it would be inefficient to amend running totals in subsequent volumes at re-run time, since every volume would have to be loaded, copied if it was on magnetic tape, and amended. To avoid the need for this, the subsequent program(s) reading the file could account for re-runs when carrying out reconciliations, provided that the difference between old and new totals is obtainable. This method does have the advantage that it does not require any extra files during the re-run. The disadvantage of using this method is that it might not be possible to ensure that the re-run is satisfactory until the next program has read the entire file and checked reconciliation totals. The amount of re-running then required to correct errors might be considered unacceptable.

When dealing with single-group volumes, it is important to ensure that a re-run does not produce a group too large to hold on one volume. This may be overcome by counting the number of bytes or words used and forcing an end of volume condition on the file during a normal run. This number should include a margin for expansion. This will work only if no program amendments affecting this counting are incorporated between the run and the re-run. This is not likely to happen during operational running, when an amendment of this nature would almost certainly mean re-running the complete file.

If this method is used for magnetic tape files, a note should be included in the operating instructions indicating the minimum length of magnetic tape reel which will be acceptable to the program, since it is sometimes the practice to cut magnetic tapes into a number of short reels for the sake of economy, or to cut off damaged portions.

MERGING BY SEPARATE PROGRAM

When several programs in a suite or job contain re-run facilities it will probably be more acceptable if just the new results from the program being re-run are written onto a new magnetic tape file at the re-run stage, instead of copying all the original results onto the file as well. The original output file and the new output file are submitted to a separate merge program which locates the original version of the re-run group, replaces it with the new version, and amends any subsequent totals in the file, producing a new output file. The advantages of this

system are that the merge program can be comparitively small and hence multiprogrammed, that it will cut program writing time becuase the merge program is only written once for several programs instead of separate merging routines being written into each program, and that the running time of the processing program is reduced because unaltered restart groups need not be copied.

This type of merge program would probably be produced for the use of more than one suite of programs. It may be a specially generated merge program or a combined sort and merge program incorporating the user's own coding which might be used to form a part of an installation's own software. This means that totals records must be designed in standardized formats and that the program may impose restrictions on the choice of block sizes. It should also be designed such that it can accept output files from both high and low level languages and it may cater for both tapes and discs. The program should report any reconciliation discrepancies encountered during the merging process.

A possible organization for this program is to code it in a modular form such that parameters submitted at the time the merge is required would call in the appropriate modules for sorting and merging tapes or discs, thus reducing the amount of coding space required in the store during the running of the program.

Use of such a program minimizes the user programming effort and standardizes merging procedures. No extra files are required during the re-run of the user program. The methods described for single-group volumes apply equally to this method. However, this method is preferable if the reconciliation in the following program is rejected, since the operators can delay the merge program until more time is available for loading and unloading tapes or discs. In such a case, the merge program could be run at a high priority with another program which has little peripheral activity.

The use of a separate program to perform the merging process is generally more favoured than other methods and is often a standard feature of an installation, rather than simply a member of a suite of programs.

RECONCILIATIONS

A further requirement of re-runs is to ensure that the original tapes and the re-run tapes have processed all the data input to the program. This means that there must be a close inter-relationship between the reconciliation system of the original run and that of the re-run.

In particular, if volume changes are to be used as restart points, the amount of data written on a re-run must be that written on the original run. This may be done by setting up a count of data characters and forcing end of volume on the original run, but it must be ensured that no program amendments which can affect the length of the controlling file have been incorporated between the original run and the re-run. It is not, in general, satisfactory to reply upon the subsequent program to check that the re-run has been successfully completed when that program reads the file, since the amount of re-running then necessary might well exceed the satisfactory level.

One suggestion is that the reconciliation information from the original run may be extracted from the next restart dump, for comparison with the new totals.

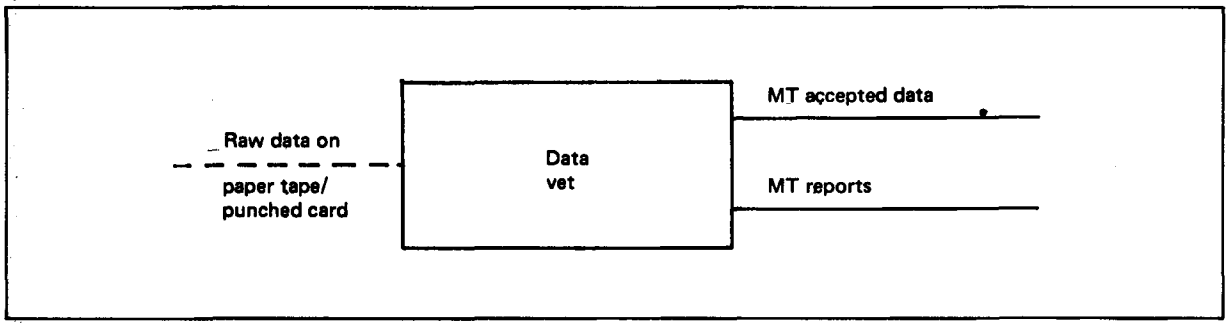
FILE CONTROL

When performing restarts and re-runs there must be stringent control of the files to prevent wrong volumes being used. Standard labelled files are uniquely identified by the file identifier, the run number, and the re-run number. In addition there is a volume number, to identify the sequence of the volumes within the particular file. These fields should all be checked before using any volume to perform restarts or re-runs.

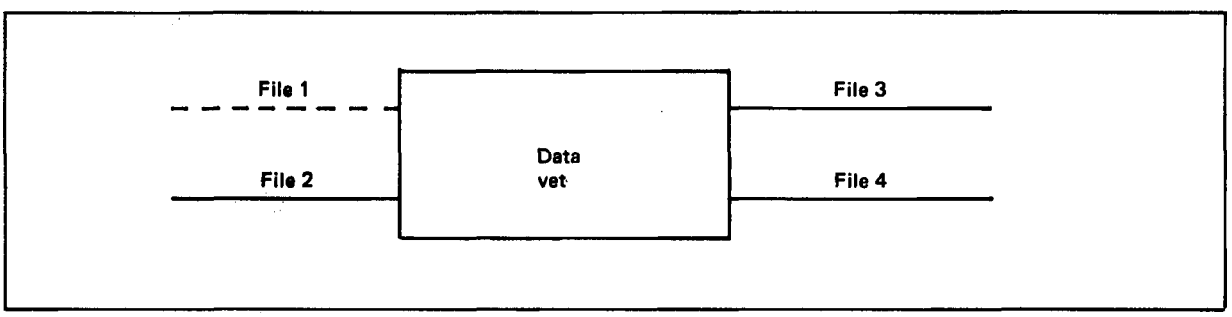
RESTARTS AND RE-RUNS BY TYPE OF PROGRAM

Data vetting programs

One of the most common types of data vetting program is represented by the top diagram overleaf. This consists of one input file, on paper tape, punched cards, or document reader containing the raw data to be vetted. The program produces two output magnetic tape files, one containing accepted data, the other containing reports on rejected data (and, possibly, totals of accepted data and so on, for off-line printing).



A more general representation of data vetting program is shown in the following diagram:



The input data to be vetted is denoted by file 1, normally on paper tape, punched cards, or document reader. The other input file, file 2, denotes a magnetic tape file for reprocessing of data rejected in earlier runs. File 3, an output magnetic tape file, contains accepted data in a form suitable for processing by the next program in the suite, usually with a sort program in between. File 4 is the reports file, containing reports on rejected data and reconciliation totals, both of which require printing. This file is normally a magnetic tape file with data in a form suitable for off-line printing when required. In some systems this file may be a paper tape file, or an on-line print file. In this case, in order to perform satisfactory restarts and re-runs, some form of heading would be required on these media to indicate which portion of the data had been processed.

SUGGESTED METHOD

The group for restarting or re-running is taken as an integral number of boxes of punched cards, reels of paper tape, or batches of documents. The exact number of boxes, reels or batches in a group partly depends on the operating circumstances, and program and system designers are advised to check with the operations manager that the grouping decided upon is acceptable. This grouping also depends upon the amount of input data to be processed for each box of cards or reel of paper tape. One possible grouping on this basis is for programs to be able to restart at each box of punched cards and at every five reels of paper tape, where a box contains approximately 2,000 cards and a reel of paper tape is about 300 feet in length and contains approximately 30,000 characters.

The restart group number should be included at the start of each reel, together with the reel number. Reel numbers should, however, be consecutive throughout the input file, and not numbered from the beginning of each group. This will satisfactorily ensure that the omission of a reel of data is detected as quickly as possible.

A log comment giving the number of the restart group which is about to be processed, may be generated by the program at the start of each restart group. This comment will enable operators to recognize which group is being processed at any time during the run. Therefore, if the program is abandoned, and a restart or re-run is necessary, the restart point will be known.

During processing a program may write the restart information onto the output magnetic tape file at the start of each restart group. This information may be produced by issuing the Checkpoint or Dump macro. On System 4 computers a user program can be used to output its own restart information in the form of an alignment block. A totals block can also be output, preceding the alignment block, giving the total number of items in the previous group and the number of items currently on the file.

TO RESTART THE PROGRAM

If the Checkpoint or Dump facility has previously been used, a restart is effected by using the appropriate software restart routines. These routines align all files and set up the program from the dumped information. Parameters must be input to specify which restart point is required.

If the Checkpoint facility is not used on System 4 computers, the type of run must be specified at the start of each run. For restarts the program should read and check any control data submitted to the run. The output file which contains the restart information should then be opened as an input file and all remaining files associated with the program are opened as for a normal run. The consistency of all these files is checked. The file containing the restart information is then run on to the appropriate restart block(s) and the restart information extracted. This is used to reset the relevant store areas, registers, etc., and to realign all the files. The restart file is then converted to its original use as an output file and processing can continue. It is essential that any failure during the restart procedures should be reported immediately and operation runs would normally be abandoned.

In some cases it may be preferable to start fresh output reels for the results of each group, (either the file containing the accepted data, the reports file, or both.) If this is done then restarting can be greatly simplified because instead of having to align the output files, new reels are opened.

TO RE-RUN THE PROGRAM

If the Checkpoint or Dump facility has previously been used re-runs are performed in the same manner as restarts.

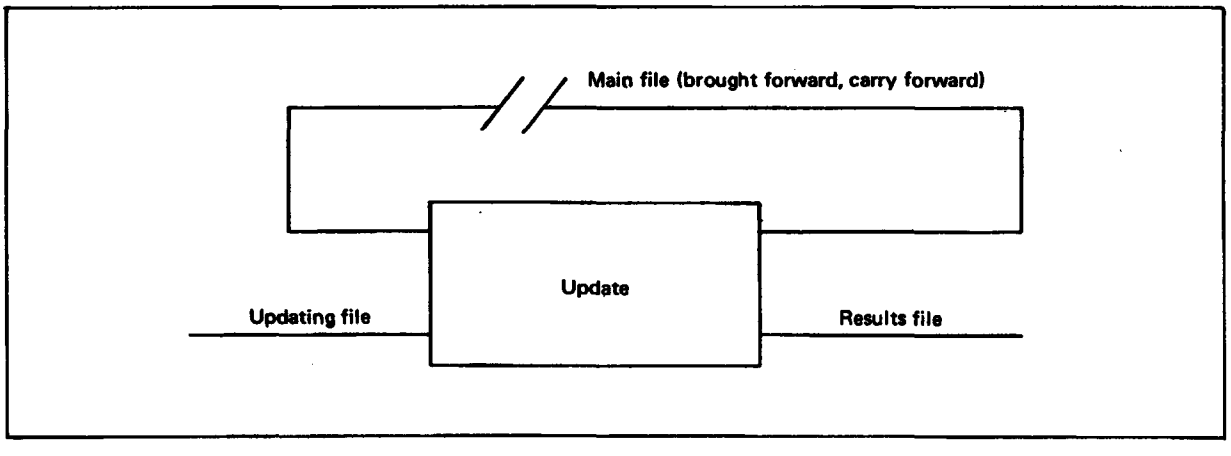
If the Checkpoint facility is not used on System 4 computers, an indication must be given to the program at the start of the run to indicate that a re-run is taking place. The associated control data is read, and checked. All files are opened, the original magnetic tape file containing the output results being opened as an input file. The consistency of these files is checked. The information from the original run held on the magnetic tape file is copied to a new magnetic tape output file, file A, until the first group requiring a re-run is reached. At this stage there are two possible methods of continuing.

1 A new output file, file B, is used for the re-run group(s) and processing continues as for a normal run using this file for output until the last group specified by the control data has been processed. The input magnetic tape file is now run on until it is also aligned at the end of the last re-run group as specified by the control data, and any remaining data on this file is copied onto the output file, file A, until the end of the input file is reached. Whilst the copying is being performed adjustments where necessary are made to any totals and alignment records as they are being written, to correct any difference in the accumulated totals. These two new tapes are then merged to form a single results file.

2 Alternatively the merging can be incorporated in the re-run. In this case file B is not needed, the re-run group(s) being processed and inserted in their correct place in file A.

In either case, it must be ensured that no data is lost or duplicated, and that reconciliation totals are correctly adjusted.

Updating programs



This process is typified by the file updating arrangement of a large multireel brought forward file which is updated to form a large multireel carried forward file; a smaller updating file which carries details of changes to be made in order to produce the carried forward file; and a smaller results file is produced which contains details of amendments made. Here it is necessary to cater for a restart and a re-run of either the main file or the subsidiary output file.

SUGGESTED METHOD

The reel of magnetic tape is regarded as the basic file unit. The reels within a file will be both a restart group and a balancing unit, and every reel of tape should hold the balance totals of the preceding reel at its head and its own balance totals at the end. It should not be assumed that a tape reel used for a re-run can hold as much data as was written on the original tape. If all reels of tape were of identical length and there were no flawed areas this condition would always exist and a re-run tape could simply replace the corrupted one. This is not so however and a method must be found of limiting the tape capacity to a practical size which will always permit direct re-run tape substitution in the original file. The recommended method is to count the amount of valid information that is being written on the carried forward output tape and to close the reel when a pre-determined limit of, say 95 per cent full capacity is reached. This count must be controlled and updated in a commercial program. Before any output to magnetic tape instruction is obeyed, control should be passed to a program routine which will check the running total (plus an allowance for the intended output instruction) against the predetermined limit. A safety margin of 5 per cent is sufficient for most variations of tape length, recording density, and for flawed areas of tape. A bigger margin may be desirable in some cases, where for instance it is thought necessary to avoid holding specific sections of data on two reels. If the section of data is equivalent to 5 per cent of tape capacity and the beginning of the section is reached immediately prior to the predetermined limit of 95 per cent tape capacity, then clearly there would be a danger of overspill. Individual project teams must therefore be responsible for deciding what limit shall be set for each multireel file.

TO RESTART THE PROGRAM

If the Checkpoint or Dump facility is used, a restart is effected by using the appropriate software restart routines.

If the Checkpoint facility is not used on System 4 computers, then on the original run alignment items and totals items are written at the start of each group, that is, at the start of each reel of the carried forward file. These items would contain an alignment item indicator, the group number, reconciliation totals from the previous reel and all details necessary to enable correct alignment of the other files in the program. Also at the end of each reel totals items are written onto the carried forward file and these contain reconciliation totals for this reel and for the file to date. Upon requiring to restart the program the control data submitted to the run is read and checked. The appropriate reels of the brought forward file and the updating file are loaded. The reel of the carried forward file at which the restart is to begin is loaded and opened as input, and the details of the alignment items are stored. The other files, including the original results file, are opened as input files and the details stored from the carried forward alignment items are used to check that the correct reel of each file has been loaded. The files are then aligned using this information and the function of the carried forward file and the results file is returned to output. Processing now continues as for a normal run, to the end of the program.

TO RE-RUN THE PROGRAM

If the Checkpoint or Dump facility is used, a re-run is effected in the same manner as restarts.

If the Checkpoint facility is not used on System 4 computers, the control data is read and checked and the required reel of the carried forward file is loaded and used to align the remaining files. This is done in the same way as for restarts, except that there are two methods of treating the results file.

1 The file is copied onto a new output magnetic tape until the required restart point is reached; this method would be suitable if the results file is small

2 If it is decided that too many decks would be used for copying, then a new results file could be opened and a subsequent program could merge the original results up to the restart point with the new results produced during the current re-run.

Whichever method is used, once the files are aligned and the function of the carried forward file is changed to output, processing continues as for a normal run until the last group specified by the control data has been processed.

At this point the reconciliation totals of the carried forward file are stored and the carried forward reel is closed. The reel with the next restart group is then loaded. The program checks the reconciliation totals held on the closed carried forward reel and those on the opened reel. If the totals are the same, the remainder of the original results file is copied to the results file and the program is closed. If the totals are not the same, then either the run must be abandoned, or the carried forward file is read to the end and all reconciliation totals held on this file from the point of divergence are adjusted according to the new totals.

SOME GENERAL COMMENTS

If any natural grouping of data on the main file occurs there are several advantages in using such groupings as restart groups. Even if such a natural grouping does not occur it may appear to be advantageous to create an artificial restart group. An example of natural grouping is to be found in a file which contains the sales records of five regions of the U.K. Each region could be called a natural group providing that it is of an appropriate size and each group is of about the same size. An example of an artifical grouping is to be found in a file which holds a range of customers numbers from 00000-09999 and which is divided so that each group contains the records of 2,000 consecutive customers, the first group being customer numbers 00000-01999, and so on. This type of grouping has the advantage of being easier to handle when programming because the alignment records will always be in the same place relative to the data on the file from one run to the next. It also means that all files can be run on to such an alignment block quite easily. Another advantage of this method of grouping is that reconciliation totals can also occur in the same position (relative to the data) on all files. If the method of grouping data is based on reels of the carried forward file, then when the carried forward file is reconciled at the end of each reel the totals for the brought forward file will almost certainly not coincide. Therefore a separate total for the brought forward file must be accumulated record by record, in order that the total is correct.

An alternative method is suggested which is of particular use when:

1 The main file is large and the time required to copy original results on to it is too great to be acceptable

2 The results file is a multireel file

3 There are several programs in the suite/job/installation and as a result a standard re-run/merge program is considered practical

When several programs in a suite or job contain re-run facilities it may be more acceptable if just the new results from the portion being re-run are written onto a new magnetic tape at the start of processing instead of copying all the original results. The original results and the new results can then be merged in a separate program. The advantages of this are that the merge program can be made small enough for multiprogramming, that the time taken to write the suite of programs is reduced because the merging routines are only written once instead of being written into every program which requires them, and that the running time of the processing program is reduced because there is no need to copy the original results. (This is particularly significant on machines with slow tape read/write speeds).

The merge program is written so that it first checks that both files are consistent, then reads data from the original file, writing the results to an output file until reaching the start of the first group which is the subject of a re-run. At this stage the program reads the results from the re-run file and writes them to the output file until the end of the re-run group is reached. Finally the results for any remaining groups are read from the input file and written to the output file until the end of the file is reached, adjusting control totals where necessary.

Main file as a disc file

If a fresh file is written every time it is updated the problems are much the same as for magnetic tape except that the restart points will not be at the start of a new volume. If the file is updated in situ any re-run must not update any record on the file which has already been updated in the original run. If a restart is required because the job was prematurely terminated or because any subsidiary output file is unreadable the following procedure is suggested if restart points are inserted:

1 Restart points are placed at regular intervals on the original run

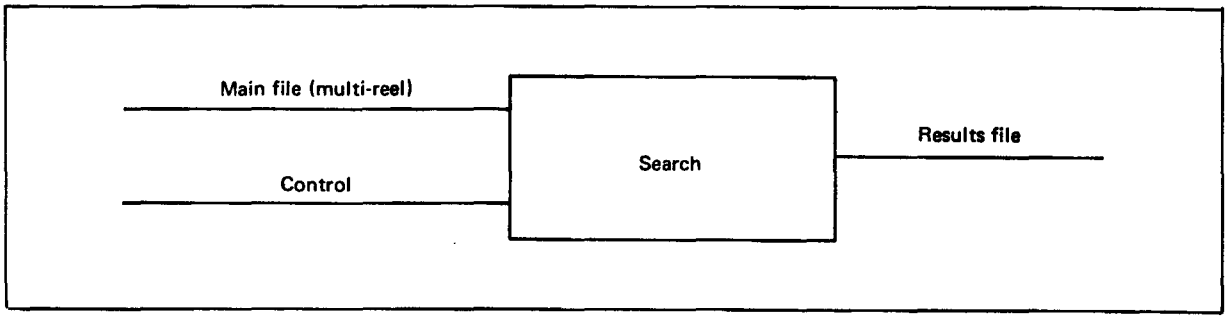
2 An indicator is placed in each block on the main file as it is updated

3 On restarting, processing is repeated from the required restart point without updating any main file record which was updated on the original run

4 Any field which is altered on the main file and whose original value may be requried on restarting is stored in additional fields on the main file record, on a separate record on the main file, or on a subsidiary file, whichever best suits the requirements of the particular job. Fields would be stored in their original form when reconciliation totals are required (if the main file is processed sequentially) or when other totals are accumulated by the program.

The main disadvantages of this system are that processing on a restart will be different from processing on the original run, requiring a substantial amount of additional coding. This is especially true if an additional file has to be matched with the main file in order to find the original value of any fields. If the main file is unreadable there is no method of resolving this without reconstituting the main file and repeating the entire run, a process which is by no means straightforward if the file has been updated in situ. If program error occurs the entire run must be repeated.

Search programs



The diagram above represents a large multireel magnetic tape file, used as an input file, which is to be searched, and from which certain specified information is to be extracted. The small subsidiary input file gives the details necessary to enable the program to locate and action the required items, and the resulting information is put on the results file for subsequent processing.

SUGGESTED METHOD

Because the input file is large, the restart group is chosen to be a reel of the input file which is being searched. The program is likely to be dominated by the time taken to read the magnetic tape file, since only minimal processing will normally be required. The normal tape passing time of a magnetic tape reel varies up to a maximum of about ten minutes per reel and this means that restart groups will often have a run time of considerably less than the suggested 10 to 20 minutes. However, this choice of restart group is desirable as it is the most convenient from an operating aspect, and is equally convenient for any additional programming required for processing the restart group.

TO RESTART THE PROGRAM

If the Checkpoint or Dump facility is used, a restart should be effected by the appropriate restart software routines.

If the Checkpoint facility is not used on System 4 computers, alignment records should be written onto the output file at the beginning of each restart group. These records should contain an alignment indicator, the restart group number, and any details necessary to enable correct realignment of the subsidiary input file, and should also contain reconciliation totals as output at the end of the previous restart group. At the end of each restart group, records should be written to the output file containing reconciliation totals for that group and for the file to date. The control data for the restart is read and checked and the required reel of the multireel search file, the subsidiary input file and the original output results file are all loaded and opened as input files. Checks for consistency should then be carried out. The results file is aligned at the end of the group which is being restarted and details of the alignment records are stored. The function of the results file is changed from input to output and the subsidiary file is aligned using the information stored from the results file. Processing then continues until the end of the run.

TO RE-RUN THE PROGRAM

If the Checkpoint or Dump facility is used, a re-run is effected in the same manner as restarts.

If the Checkpoint facility is not used on System 4 computers, alignment records should be written to the output file, at the start of each restart group as for a restart. Similarly reconciliation totals should be written at the end of each group. The control data for the re-run is read and checked. The first reel of the search file to be re-run is loaded and checked for consistency. The original results file is opened as an input file and the information in it is copied onto a new output magnetic tape results file until the first group to be re-run is reached. The details of the alignment records at this point are then stored. From these details the subsidiary input file is aligned. Processing continues as if it were a normal run until the last group specified by the control data has been processed. The original results file is now run on until it is also aligned at the end of the last specified group. The data remaining on this file is copied onto the new results file until the end of the file is reached. Whilst the copying is being done adjustments are made to the totals and alignment blocks as they are being written, to correct any differences in the accumulated totals.

SOME GENERAL COMMENTS

The method of selecting groups on a search file are those considered for an update file, the difference being that search programs use a multireel input file, and update programs use a multireel output file.

Edit programs

The basic function of an edit program is to read information from an input file and to write it again in a re-arranged order on an output file. As a consequence of this either the input and output files may be approximately equal in length, or one of the files may be considerably longer than the other.

In the first case the situation is approximately analogous to the updating program, and a similar method of restarting and re-running can be used. In the second case the configuration is more like that used in search programs and so the method used for search programs would be more suitable. In either case, however, as edit programs are often calculation dominated it will almost certainly be desirable to choose a natural grouping as the basis for determining the restart points. For example, the result types on the output file might be used if there was about the same amount of time spent for each type.

Print programs

A restart technique that can be used with print programs involves submitting a special parameter at the beginning of the restart. This identifies the items to be printed. The program suppresses printing until the required item is found and then printing commences. There are normally four types of printing which may be required at a restart:

1 To print a specified result type or types

2 To print a specified sheet or sheets

3 To print from the start of a specified result type, to the normal end or run

4 To print from a specified sheet, to the normal end of run

Any print programs which are not standard should be designed to cater for these requirements upon restarting. In some cases type 4 may have to be specified as a combination of 2 and 3.

Calculation programs

In all the programs which have been considered the factor determining run time has normally been dependent on the peripheral speed. Usually there is a natural break in the files which coincides conveniently with the processing interval required by the group.

With calculation dominated programs, however, there is a different problem. The quantity of input and/or output data may be very small and yet because of the length of time required to perform calculations the program may still have to be divided into reprocessing groups. Calculation programs may be classified as one of two types:

1 Processing consists of a series of similar loops, e.g. a standard costing program may have to perform the same large calculation loop for each of about 100 materials

2 Processing consists of only one large calculation, which does not loop as described for type 1

In the first case a division of processing time is not difficult to establish. The average time taken for one loop can be found and when a number of loops have been performed in the time allotted to a group, any information which would be required to begin reprocessing at that point can be dumped onto the results file after a group mark. It is also probably necessary to communicate with the operator at this point and to give him sufficient information to enable him to compile control data when it is required. To restart or re-run the program the files can be re-aligned and the store re-set from this information.

In the second case, however, each program must be considered according to its individual features.

Punched card or paper tape output programs

There would appear to be three main problems in programs with punched card or paper tape output only. The first problem is to define the groups for the operators. The group division need not necessarily be dependent on the quantity of the output results produced. An alignment block on the input file may solve the problem if a suitable grouping can be determined.

The second problem is to find somewhere to dump the information which is necessary to reset the store when the program is restarted or re-run. The best solution is to set up a disc file solely for holding restart information.

The third problem is control of the results of the restarted or re-run program. A system must be devised to prevent the results produced from a restart or re-run getting mixed up with those produced during the previous runs.

Appendix 9 Programming estimating

Each stage of estimating is at a progressively more detailed level. The initial stage of programming estimates is to establish the work content at project level. These estimates are then expanded at program level to give a detailed work content in terms of effort and time. The estimator must be rigorous in ensuring that there is sufficient information available, and that the task has been broken down into sufficient detail, to estimate the following factors:

1 The number of instructions that will be required to program the task

2 The accuracy of the estimate of instructions. This is a useful self-assessment of confidence and completeness of the understanding of the problem

3 The complexity of the logic and functions to be performed by the instructions, in relation to a nominal average complexity for the type of task. This is known as the complexity factor

It is possible that the estimate of number of instructions will be made for a language different to that to be used for implementation. It is necessary, therefore, to have factors which can be used to convert from the estimating language to the implementation language.

The figures given in Table 1, at the end of this appendix, reflect the power of the language to perform an average programming task, and can therefore be used for converting numbers of instructions from one language to another. The figures do not relate to time estimates and must not be used for time conversions, the reason being that certain tasks take the same time irrespective of language.

The estimated numbers of instructions should be converted to the appropriate language and the resulting number of instructions should be used for establishing the work content.

The number of instructions in this context are defined as follows:

PLAN instructions

For estimating purposes these do not include directives, Upper or Lower variable and preset definitions, or parameter constants for subroutines. Macro instructions are counted as written, not as expanded, and each distribution line is counted as one instruction.

Usercode instructions

For estimating purposes these do not include translator instructions, file definitions, or definitions of constants and work areas (DC or DS). Macro instructions are counted as one instruction and not as expanded.

COBOL statements

For estimating purposes each verb in the Procedure division is counted as an instruction. IF or IF/ELSE are taken to be a verb in this context but NOTE statements are not included.

The reason for these definitions is that only the active instructions are considered when size estimates are being made. The time estimates allow for the overhead of writing data definitions etc.

PROJECT ESTIMATES

Project man-effort

It is necessary to be able to make estimates of the total man-effort required to implement a project, that is from the start of writing program specifications to the end of system testing. Such estimates are obviously not very accurate since there will not be much detailed information available, and the work content can be taken to be equivalent to the number of instructions since any complexity factors are likely to average out over the suite.

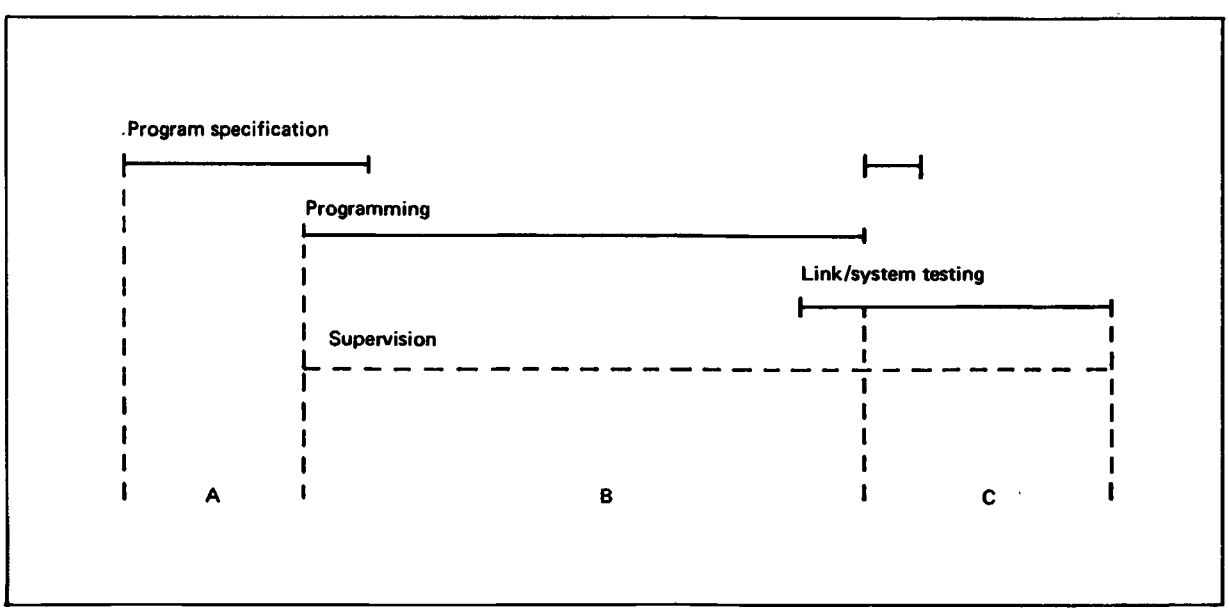
The figures given in Table 2, at the end of this appendix, are intended for guidance and show the percentage of effort required for supervision, specifying, programming and link/system testing, and also give suggested average man-days effort per 1,000 work content units for each of these tasks.

Additionally, as a quick method of obtaining man-effort estimates, factors are given for converting the work content direct to total man-effort for project implementation. The resulting man-effort figures are for a 'normal' project, that is one with straightforward processing and file structures, and not requiring any particular new techniques. Projects which vary from this norm will require more man-effort and the table below gives some guidance on the adjustment required for different types of project. The estimator should,however, seek the advice of more experienced staff in establishing any necessary adjustment.

Type of project	*Normal project effort multiplying factor*
Normal	1.0
Increasing complexity of processing and file structures. Variety of types of data. Changing requirements.	1.5
Large variety of types of data. Critical running time scale.	2.0
New techniques (for example, real-time)	2.5

Project elapsed time

It is not possible to lay down standards for project elapsed time since this is dependent upon the number of staff used. However, it is possible to give guidance in some areas. A bar chart of the time structure of a project, showing overlapping tasks, is given below:



The elapsed time can be roughly calculated as the total of (A + B + C) + 25%

where $A = \frac{75\% \text{ of program specification man-days}}{\text{Average number of staff used on stage}}$

$B = \frac{100\% \text{ of detailed programming man-days}}{\text{Average number of staff used on stage (excluding supervisory staff)}}$

$C = \frac{75\% \text{ of link/system testing man-days}}{\text{Average number of staff used on stage (excluding supervisory staff)}}$

The additional 25% is to cover holidays, sickness and any other non-productive days. An extra 10% to 30% should be allowed to cover estimating errors and possible trial turn-round delays.

Years/weeks conversions

Care must be taken when converting man-days effort, or elapsed days, to weeks and years. Assuming a five-day week, there are 260 working days in a year, that is 52 weeks of five days. After deducting non-productive days there are about 208 effective man-days per year, that is 52 weeks of four days. Man-days effort therefore should always be converted on the basis of 260 days. Elapsed time should also be converted on this basis since a 25% allowance has been included. Obviously if elapsed time is converted on a basis of 208 days then the 25% allowance is not necessary.

Team sizes

There is an optimum team size for any project giving what may be considered as 'normal' elapsed time. A suggested method of estimating the optimum size is to assume one team member per 3000 PLAN or 2500 Usercode work content units, or per 1700 COMPACT COBOL units or per 1000 FULL COBOL units. This will give an optimum team size including supervisory effort. If a project is required in less than normal elapsed time then more man-effort must be expended.

There is no direct relationship between the reduction in the time scale and the extra man-effort required to achieve this reduction.

Example

Project time scale required = $^2/_3$ of normal time

Project man-effort required = 2 x effort for normal time

Project time scale required = $^1/_2$ of normal time

Project man-effort required = $^4/_5$ x effort for normal time

Any assessment of this increase in man-effort must rely on experience of past projects and the advice of more experienced staff should be sought in such cases.

PROGRAM ESTIMATES

The previous sections consider methods for obtaining broad estimates of effort and time for an entire project. For the detailed programming stage, however, it is necessary to make detailed estimates for each individual program.

Program work content

It is necessary to establish the work content of a program in such a way that it can be used as a basis for calculating man-effort, and can also provide a means of measuring the accuracy of the estimate by comparison with actual results.

Work content is calculated as follows:

Work content (units) = number of instructions x complexity factor

It is recommended that work content estimates should be carried out by two people as a means of establishing their reliability and that results should be compared with similar projects whenever possible.

Number of instructions

It is possible for a very experienced estimator to assess the number of instructions merely by studying the problem. Very few staff, however, have the experience to use such a method and it is recommended that a rough flowchart is drawn and each box is assessed separately. If an outline chart is available this should be used instead of the rough chart.

Complexity factors

The complexity of the processing must also be taken into account since it is possible that two programs can contain the same number of instructions but one of them can be more difficult than the other and thus have a higher work content. The factors which are listed are intended for guidance in establishing the types of program

complexities. If a program is likely to be too large for the store, and will therefore require overlaying, the complexity factor should be raised.

Type of program	*Complexity factor*
SIMPLE Extracts of one type of data from one file Prints of one format from one file Data vets of one type of data	0.75
AVERAGE Data vets of several types of data Updates of simple structure files by a few types of current data Prints with several formats	1.0
COMPLEX Data vets with many data types and extensive consistency checking Updates of more complex structure files with several types of current data Prints on pre-printed numbered stationery including updating	1.25
VERY COMPLEX Complex updates with high degree accuracy calculations, or requiring critical run times, or with many types of current data	1.5
Real-time processing	2.5 to 3.0

Program man-effort

The work content is converted to man-effort by using the figures given in Table 3, at the end of this appendix. The figures give the average man-effort required for 1,000 units of work content of a fully documented program or module. However, where a program is divided into modules so that several programmes can be employed, and the modules are grouped into segments for testing purposes, time should be allowed for defining the modules and linking the segments.

For pre-logic and checking allow ½ to 1½ days per module.
For post-independent trials allow ½ to 1½ days per segment.

PROGRAMMER VARIATIONS

The man-effort required will vary according to the experience and ability of the programmer. The figures given in Table 3 are applicable to an average programmer of about 6 to 9 months experience and will require adjustment according to the factors shown in the table below. Assessing the ability of a programmer to carry out a particular task is very subjective and must be the responsibility of the programming supervisor. One method suggested for estimating purposes is to relate ability and experience so that the same adjustment factors can be used. For example, a programmer may have 12 months experience but be of less than average ability, and therefore be rated as having 9 months experience.

Experience (months)	*Effort adjustment factor*
0–3	1.4
3–6	1.2
6–9	1.0
9–12	0.9
12–15	0.8
15–18	0.7
18–21	0.6
21–24	0.5

Program elapsed time

Before any planning of resources can be carried out it is necessary to calculate the elapsed time required for each program or module. This is based on the man-effort required but there are many other factors which affect the conversion of man-effort to elapsed time:

Major holidays

Formal training

Sickness, minor holidays, informal training, meetings

Unschedule maintenance, minor system changes, minor estimating errors

Lack of available trial time

Number of other activities with which the programmer is concerned

Working environment

These factors can be grouped so that ways of making allowance for them can be given.

PREDICTABLE OVERHEADS

These include major holidays and formal training courses, that is any overhead likely to last for a period of two to three weeks. Such overheads should be known well in advance and can therefore be included as the exact amount of elapsed time.

UNPREDICTABLE OVERHEADS AND ACTIVITIES

It is recommended that 25% of man-effort should be included to cover those overheads which cannot be predicted. The 25% of man-effort must be absorbed into the program implementation time to make control more realistic, and it is therefore suggested that a five-day week is considered as four effective days plus one unproductive day, that is 25% of effective time. The following calculation is then valid.

$$\text{Elapsed time (weeks)} = \frac{\text{man-effort (days)}}{4}$$

PART-TIME WORKING

Part-time working may arise when there is insufficient trial time available, and the programmer is not fully occupied during the trial activities, or when the programmer is engaged on other known activities. It is necessary to access to what degree the programmer is occupied for each of the programming stages. This is calculated in the following way:

$$\text{Elapsed time (weeks)} = \frac{\text{man-effort (days)}}{x}$$

where

man-effort is the number of days for each programming stage.

x is the proportion of effective available days.

SUMMARY

Elapsed time is calculated as follows:

$$\text{Elapsed time} = \text{scheduled holiday (weeks)} + \text{scheduled training (weeks)} + \frac{\text{man-effort (days)}}{x}$$

where

x is 4 for full-time working
3 for threequarter-time working
2 for half-time working
1 for quarter-time working

TRIALS ESTIMATES

Program trials are required to produce a working program which satisfies the programmer. This can be considered in three stages:

Initial compilation

Program or routine testing

Linking of routines to form a program

Initial compilation

The initial compilation stage is allowed for obtaining a usable object program and three trials should be allowed regardless of size of program, language and experience.

Program or routine testing

In this context a program trial is defined as a compilation plus a trial of the program or routine. The number of program trials required is based on the work content and it is recommended that 10 program trials are allowed per 1,000 PLAN units and 12 program trials are allowed per 1,000 units for other languages; however, the number of trials should never be estimated as less than 2.

Linking of routines

This stage is only required when a program is written as several routines and these routines are grouped into segments of one or more routines which are tested independently.

Up to three program trials should be allowed for each linking together of segments. For example, two segments tested independently and then linked together should be allowed up to three trials; three segments tested independently and then all linked together should be allowed up to six trials; four segments tested independently, linked into two segments, and finally into one program should be allowed up to nine trials.

Summary

The number of trials is calculated as follows:

Number of productive trials = 3
+ 12 per 1,000 work content units
(or 10 if using PLAN)
+ up to 3 per segment linkage.

It is recommended that for estimating purposes 25% is allowed for non-productive trials, that is trials which do not allow any progress to be made due to programmers' errors in submission. The figure given to the individual programmer as a target should always be the estimated number of productive trials, that is the number of trials excluding the 25% for non-productive trials.

NUMBER OF PROGRAM TURN-ROUNDS

Computer time availability normally dictates the number of turn-rounds, that is the number of submissions of a program or routine, which can be achieved per week. However it is still necessary to ensure that the number of turn-rounds does not exceed the man-effort required to analyze the trials. The average man-effort can be calculated as follows:

$$\text{Man-effort per trial} = \frac{\text{man-days per trial stage}}{\text{estimated number of productive trials}}$$

The average number of turn-rounds per week can then be calculated by dividing the man-effort available per week by the average effort per trial. This figure can then be balanced against the amount of available computer time.

COMPUTER TIME USAGE FOR PROGRAM TRIALS

There are many factors which are likely to affect the usage of computer time for program trials:

Number of instructions in programs

Operating system

Computer/configuration

Language

Options selected when compiling

Type of program

Volume of data processed

Diagnostics requested

Due to the variety of these factors it is not possible to give a detailed method of estimating computer time per compilation and trial since the calculation would be too time consuming; moreover at the estimating stage some of the factors are not known. It is suggested that users record their own timings and use these to estimate future timings.

It is important to remember that computer time is needed for setting up trial data files and an allowance must be included for this. The calculation for trial time is as follows:

Trial time = 3 x time per compilation (initial compilations)

+ number of program trials (including 25% allowance) x time per program trial

+ ½ to 1 hour (trial data set-up)

LINK/SYSTEM TRIALS

Standards cannot be given for the number of trials or for computer usage for link and systems trials since these depend mainly on the volume of data and the number of sets of data to be processed. It is, however, possible to give a few rough guides which may be helpful when making estimates at an early stage before a detailed plan for testing has been established.

Link trials

The number of trials is more likely to depend on the number of programs and connecting files than on the number of instructions. It is suggested that a minimum of three trials per program is allowed.

System trials

The number of trials will depend very much on the accuracy of the systems specification and the number of system changes which have occurred during programming. Here again it is only possible to suggest a minimum figure and it is likely that this will be about two trials per program per set of data processed.

Computer time usage

The computer time usage will depend on the number of times programs have to be recompiled and on the volume of data processed. It is suggested that the time used for the combined stages of link and system trials will be of the same order as for the program trials.

LANGUAGE EXPANSIONS

System 4

The figures given in Table 4, at the end of this appendix, show the average instruction/statement sizes per program. These averages are not applicable to individual small modules, since the variations on small units of coding can be very large.

The language expansions in the table include any overheads caused directly by the statements, but exclude the following:

FCP

DTFSR tables

I/O areas

Data areas

Constants

Compiler overheads

1900 Series

Because of the nature of the 1900 Series processing, no figures can be given for the average instruction/statement sizes per program. However for programs written in PLAN, a rough guide may be 1 word per instruction with an allowance for macro expansions. It is suggested that users establish the average number of macro expansions per program, and use this as a factor for estimating the total number of instructions.

For COBOL, no such guide can be given, and programmers should use their experience to gauge the instruction size.

TABLE 1

POWER OF LANGUAGE CONVERSION

The factors given below should be used for converting the number of instructions from one language to another.

		Writing language			
		PLAN	Usercode	COMPACT COBOL	FULL COBOL
Estimating language	PLAN	1.0	0.7	0.5	0.3
	Usercode	1.4	1.0	0.7	0.4
	COMPACT COBOL	2.0	1.5	1.0	0.6
	FULL COBOL	3.0	2.5	1.7	1.0

To obtain the appropriate conversion factor take the figure at the intersection of the Estimating language line and the Writing language column.

Example

A program has been estimated as 1,500 Usercode instructions but is to be written in FULL COBOL. The intersection of the Usercode line with the FULL COBOL column gives a factor of 0.4. The number of FULL COBOL statements will therefore be 1,500 x 0.4 = 600 statements.

TABLE 2

PROJECT MAN-EFFORT TABLES

The table below shows the percentage of man-effort likely to be used on various tasks, and also the number of man-days effort required per 1,000 work content units in each given language.

	PLAN		Usercode		COMPACT COBOL		FULL COBOL	
	% Effort	Man days (1K)	% Effort	Man days (1K)	% Effort	Man days (1K)	% Effort	Man days (1K)
Supervision	10%	6	10%	8½	10%	11	10%	14
Specification	9%	6	10%	8½	11%	12	15%	21
Detailed programming	56%	36	55%	45	54%	60	50%	70
Link/system testing	25%	16	25%	20	25%	28	25%	35
Total	100%	64	100%	82	100%	111	100%	140

A quick method of estimating total effort is as follows:

$$\text{Total man-days effort} = \frac{\text{number of work content units}}{x}$$

where x = 15 (PLAN)
12 (Usercode)
9 (COMPACT COBOL)
7 (FULL COBOL)

The table below gives a comparison of the man-effort required to implement the same job in each of the different languages.

	PLAN	Usercode	COMPACT COBOL	FULL COBOL
	Man-days for 14,000 work units	Man-days for 10,000 work units	Man-days for 7,000 work units	Man-days for 4,000 work units
Supervision	90	85	78	55
Specification	85	85	85	85
Detailed programming	500	450	420	280
Link/system testing	225	200	192	140
Total	900	820	775	560

TABLE 3

PROGRAM MAN-EFFORT TABLES

The table below shows the man-days effort required for a fully documented program of 1,000 work content units.

	PLAN	Usercode	COMPACT COBOL	FULL COBOL
Logic plus checking	5	6	10	15
Coding plus checking	12	16	20	20
Trial data preparation	4	5	7	9
Initial compile plus dry-run	3	3	5	6
Trials	12	15	18	20
Total	36	45	60	70

To convert these figures to the effort for a different number of instructions it is only necessary to multiply by the work content units divided by 1,000.

Example

500 FULL COBOL work units = $70 \times \frac{500}{1000} = 35$ days

The figures are not valid for less than about 40 PLAN, 100 Usercode, 80 COMPACT COBOL, and 70 FULL COBOL work units, that is about 5 days man-effort.

The table below gives a comparison of the man-effort required to implement the same program in each of the different languages.

	PLAN	Usercode	COMPACT COBOL	FULL COBOL
Program work content units	1400	1000	700	400
Total man-days	50	45	40	28

TABLE 4

SYSTEM 4 INSTRUCTION SIZES

Language	*Average instruction/statement size*
4–30 Usercode	5.1 bytes *
4–40, 50, 70 Usercode (mainly binary)	4.2 bytes *
4–40, 50, 70 Usercode (mainly decimal)	4.5 bytes *
4–30 COBOL	15–20 bytes (average instruction programs) 20–25 bytes (complex instruction programs, for example, extensive subscripting, many report items)
4–40, 50, 70 COBOL	12–18 bytes (average instruction programs) 18–24 bytes (very complex instruction programs, for example, extensive subscripting, use of compute)
CLEO	15–20 bytes (average instruction programs) 25–30 bytes (very complex instruction programs)

* These figures do not include macro expansions. It is suggested that 0.5 bytes should be added per instruction to allow for these expansions.

Appendix 10 Programming work control scheme

The recommended control scheme consists of a number of forms on which the detailed information needed for control can be recorded. There are nine forms which have been designed so that sub-sets can be selected according to the level of control required. Four different levels are suggested, each adding to the previous level and giving a higher degree of control; however, other combinations of the forms are possible. For instance Level 1 plus Level 3 may also be used to advantage.

SUMMARY OF FORMS

The table below gives a summary of the forms required for each level, and also when, and by whom, the forms should be used:

Level	*Form*	*Entry of estimates*		*Update by Actual*	*Responsible*
		Initial	*Revision*		
Level 1 Basic control	Project check list			As necessary	Supervisor
	Program check list	✓	✓	As necessary	Supervisor/ programmer
	Programming schedule	✓	✓	Weekly	Supervisor
Level 2 Level-1 plus formal reporting	Project progress report			Weekly	Supervisor
	Program progress report			Weekly	Supervisor
Level 3 Level 2 plus stricter control and basic statistics	Program record sheet	✓	✓	Weekly	Supervisor/ programmer
	Trials record sheet	✓	✓	Weekly	Supervisor/ programmer
Level 4 Level 3 plus full statistics	Project history record	✓	✓	End of project	Supervisor
	Program history record	✓	✓	End of program	Supervisor/ programmer

LEVEL 1 DETAILS

Level 1 is the minimum level of control required by an installation, and gives a basic control using the minimum of effort. It allows recording of estimates, recording of the plan, checks that tasks are performed and recording of actual against estimated progress. The way in which the forms are used is described below, and examples are included at the end of this appendix.

Project check list (Example 1)

This form gives a list of major tasks which are likely to be necessary for any project. The tasks are presented approximately in the sequence of performance, and the list is intended to act as a reminder of jobs which are outstanding.

The form is updated as tasks are started and completed. Columns are provided for entering the start and completion dates of each task thus giving a record that the tasks have been performed and also of the elapsed time taken. Where a task is not applicable to the project, N/A should be entered to show that it has not been overlooked. Space is provided at the bottom of the form for recording the completion of the design tasks for individual programs.

Program check list (Example 2)

This form allows recording of program estimates and also gives a list of the tasks which must be carried out for any program. The tasks are in the sequence of performance, and the list is intended to act as a reminder of jobs which are outstanding.

The initial program estimates are entered in the upper section of the form which serves as a record of the estimates and also shows the programmer what effort is expected. Space is allowed for three revisions to the estimates. The form is updated by the programmer as each task is completed. Columns are provided for date and signature of both author and checker thus giving a record that the tasks have been carried out, and the work checked where necessary. Space is also provided for recording the completion of a major systems change.

Programming schedule (Example 3)

This form is provided as a scheduling aid and a means of recording the final implementation plan in bar chart form. Once a satisfactory allocation of programs to programmers has been established the plan should be recorded showing programs, and the programmer concerned, on the left of the form and time in weeks across the top. The bar chart lines represent estimated elapsed time, that is each week is considered as four days man-effort, and are marked to show the time for each detailed programming stage. Dotted lines are used to denote part-time working.

The chart is updated for each program each week so that actual progress can be compared with estimated progress:

1 The work remaining on the current stage is assessed and a mark is made on the bar chart estimate line to show this. The marked position is compared with the actual time position across the top of the bar chart to obtain the progress information. In Example 3, for instance, at the end of week 1 (21/6) no work had been done on the data vet and it was therefore one week behind schedule; at the end of week 3 (5/7) there was one day of man-effort required to complete the logic stage and the data vet was therefore two days behind schedule; at the end of week 7 there were two days effort remaining to complete the trial data and the data vet was therefore two days ahead of schedule.

2 A record of actual effort can be kept by drawing a line to show the man-effort expended each week. This record will not be absolutely accurate but will give a rough idea of effort expended.

   Lines can be drawn to link the actual effort line to the estimate to give a pictorial view of progress.

   The chart must be changed, or redrawn, if the estimates are revised so that the measurement of progress is realistic (see Print program in Example 3).

LEVEL 2 DETAILS

Level 2 adds formal reporting to the basic control given by Level 1. Forms are provided so that reports are made in a standardized format. A description of the reports is given below and examples are included at the end of this appendix.

Project progress report (Example 4)

This form is provided for reporting on progress on the project as a whole. A list of content headings is included to act as a reminder of the subjects which should be considered when reporting.

A report should be written each week, with the intention of conveying an overall picture of the progress in the programming of the project. Ticks should be entered in the check list boxes for all subjects included in the report, and the subject matter should cover the current position, special actions taken, any future difficulties which can be predicted and recommended action to ease or avoid difficulties.

Program progress report (Example 5)

This form is provided for reporting progress on each individual program.

A report should be made each week showing for each program in the suite the current progress position. This information can be obtained from the bar chart. Explanations of delays, use of resources and suggestions for corrective action should be included.

LEVEL 3 DETAILS

Level 3 adds to the lower levels a more exact method of recording actual effort and computer time used, thus giving an improved comparison of actual and estimates, tighter control and some basic statistics. The way in which the forms are used is described below and examples are included at the end of this appendix.

Program record sheet (Examples 6 and 7)

This form is provided for recording program estimates, actual effort expended, and the current progress position for each program. One form is used for each program or independent routine.

The initial entries are made as follows:

1 The start and end dates are entered from the programming schedule

2 The man-effort estimates for each detailed stage are entered in the first line of the form, and the total man-effort in the Actual and Expected man-effort remaining columns

3 The number of man-days expected per week is entered in the Notes column, that is, the figure used for converting man-effort days to elapsed weeks

Updating entries are made each week, one the scheduled start date has been reached or work has commenced:

1 The week ending date, names of programmers concerned, and actual effort in days are entered in the appropriate columns. More than one line can be used if several programmers are concerned

2 The effort required to complete the current stage is assessed and entered in the Notes column. If the stage has been completed a tick is entered in the column for that stage. The effort remaining is not the estimate minus the actual effort, but is assessed according to the outstanding tasks. For example, if half the work is remaining and ten days have already been used then ten more days effort will be required, whatever the estimate

3 The entries for the other three columns are calculated:

Actual man-effort remaining = assessed days remaining on current stage + estimated days for future stages
Example

In Example 6, week ending 26/7

Actual man-effort remaining = 1 day (current coding stage)
+ 6 + 5 + 15 days
= 27 days

Expected man-effort remaining = previous week's entry - expected days per week
Example

In Example 6, week ending 26/7

Expected man-effort remaining = 32-4 days
= 28 days

Days ahead/behind = expected man-effort remaining - actual man-effort remaining
Example

In Example 6, week ending 26/7

Days ahead/behind = 28-27
= 1+ days (ahead of schedule)

Effort used on stages already completed, due to system changes can easily be recorded, (see Example 6, week ending 6/9)

When the program has been completed the actual effort figures are totalled and the form provides a record of the progress throughout implementation.

Revisions to estimates can also be made with very little difficulty. This is best described by reference to Example 7, Estimate 2. The program man-effort was re-estimated and the revised standard man-effort estimates entered on the next available line. Since the schedule was also revised the actual and expected effort remaining were both 13 days, rather than the expected effort being 40 days.

The Programming schedule form is updated and revised more easily than in Level 1 since all the actual figures are recorded on the Program record sheet. All that is necessary to get an overall picture of the project is to mark off the days ahead or behind the current date. Example 8 shows how this is done; for instance in the week ending 26/7 the data vet was 1 day ahead according to the Program record sheet. Care must be taken to keep the Program record in line with the schedule. If holidays are included on the bar chart then the expected man-effort remaining is not reduced; and for unscheduled holidays the expected man-effort remaining is reduced.

Trials record sheet (Examples 9 and 10)

This form is provided for recording the actual trial time used and comparing actual against estimated usage. One form is used for each program or independent routine.

The initial estimates are entered on the first line of the form and show the time for trial data set-up separately from the compilation and trial time (see Example 9).

Updating entries are made each week that computer time is used. The number of machine runs to set up data, and the time used, are entered for the week and added to the cumulative total for comparison with the estimate. For each week the number of runs and time used, are entered separately for initial compilations and program trials (compile and trial). These entries are totalled for the week, and the weekly total is added to a cumulative total for comparison with the estimate.

At the end of the program trials stage the weekly columns are totalled to give a breakdown of the trial usage into initial compilations and program trials (compile and trial).

Revisions to estimates can be made by entering the new estimates on the next available line (see Example 10).

LEVEL 4 DETAILS

Level 4 adds summarized statistics to the details given by lower levels, and therefore provides for setting up a library which can be used as a reference when estimating for new projects and also for improving the program estimating standards. The use of the forms is described below and examples are included at the end of this appendix.

Project history record (Example 11)

This form is provided for summarizing the project details, including time taken and effort used.

The initial entries are made at the start of programming work on the project and include basic details of the computer and language to be used. The initial project estimates are also recorded. During the project the only entries required are revisions to these estimates.

When the project is completed the actual figures are recorded on the lower part of the form, including start and end dates of stages, computer time usage, number of staff and effort used, and types of programs written.

Program history record (Example 12)

This form is provided for summarizing the implementation details of a program, and one form is used for each program. The initial program estimates are entered in the upper part of the form. During the project the only entries required are revisions to these estimates.

When the programming work has been completed for a program the actual implementation details are entered, including program size and work content details, man-effort and elapsed time, and computer usage. Most of these figures are taken directly from the totals entered on the Program and Trials record sheets. Additional information on the type of processing carried out by the program is entered on the reverse side of the form.

EXAMPLES OF FORMS

The following examples show how the entries are made on the forms. The entries made in the form examples are from a single project and are therefore consistent throughout.

Example	*Form name*	*Level*
1	Project Check List	1
2	Program Check List	
3	Programming Schedule	
4	Project Progress Report	2
5	Program Progress Report	
6	Program Record Sheet	3
7	Program Record Sheet	
8	Programming Schedule	
9	Trials Record Sheet	
10	Trials Record Sheet	
11	Project History Record	4
12	Program History Record	

International Computers Limited Data processing **Project check list** **ICL**

Project controller	J. SMITH	User	XYZ SERVICES LTD.
Systems contact	K. BROWN		
User contact	N. HARRIS (C.P.)	Project	PROJECT ANALYSIS
Chief programmer	L. JONES		
Senior programmer	M. ROBINSON		

Activity

A. Initial negotiations	Start	Complete
1. Request for project		
2. Produce broad estimates		
3. Agree main language		
4. Obtain authorisation to proceed		
5. Agree charging method		
6. Produce preliminary time-table		
7. Arrange trial time	Not Applicable	
8. Make preliminary staff request		

C. Linked/systems trials	Start	Complete
1. Produce operating instructions		
2. Plan linked trials strategy		
3. Produce linked trial data		
4. Run linked trials		
5. Plan systems trials strategy		
6. Ensure systems data produced		
7. Run systems trials		
8. Revise run-time estimates		

B. Programming preparation		
1. Vet systems specification	N/A	
2. Vet or produce outline specification	2.4.69	11.4.69
3. Revise estimates, schedule and staffing	1.5.69	10.5.69
4. Produce program specifications	2.4.69	10.5.69
5. Produce outline flowcharts	2.4.69	10.5.69
6. Revise schedule and produce control forms	9.5.69	10.5.69
7. Revise trial arrangements		
8. Produce run-time estimates		
9. Obtain staff and discuss project		
10. Plan trial strategy for each program	12.5.69	19.5.69
11. Appoint trials controller	N/A	

D. Operational preparation	Start	Complete
1. Arrange data-prep facilities		
2. Arrange computer time		
3. Run take-on programs		
4. Parallel running		
5. Obtain user acceptance of programs		
6. Finalise all documentation		
7. Arrange retention of media		
8. Release other media		
9. Hand over project		
10. Define maintenance responsibilities		
11. Produce project statistics		

Program tasks

Program name	Specify	Outline chart	Trial plan
CARD TO TAPE	✓	✓	✓
DATA VET	✓	✓	✓
UPDATE	✓	✓	✓
EXTRACT	✓		✓
PRINT	✓	✓	✓

Program name	Specify	Outline chart	Trial plan

FORM 14/109/1 (12.69)

Example 1

International Computers Limited Data processing **Program check list** **ICL**

Program number A002A Program name DATA VET

Programmer A.N. OTHER

Estimates

	Estimate no.	Estimate no.	Estimate no.	Estimate no.
Number of data statements		420		
Number of procedure statements/ instructions	700	770		
Complexity factor	1	1		
Work content units	700	770		
Experience factor	1	1		
Total store required	22,000	22,000		
Man-effort (days)				
A Logic and checking	11	11		
B Coding and checking	15	15		
C Trial data	6	6		
D Initial compilation and dry-run [number]	5 (3)	5 (3)		
E Trials [number]	15 (9)	15 (9)		

Implementation

Activity	Program Author Date	Program Author Signature	Program Checker Date	Program Checker Signature	Major system revisions Author Date	Major system revisions Author Signature	Major system revisions Checker Date	Major system revisions Checker Signature
A. Logic and checking:-								
1. Study specification	27/6	ANO						
2. Outline flowchart draw/check			28/6	ANO				
3. Detailed flowcharts draw/check	11/7	ANO	12/7	AS	3/9	ANO	3/9	AS
4. Check estimates	12/7	ANO						
B. Coding and checking:-								
1. Code and check	19/7	ANO	26/7	AS	4/9	ANO	4/9	AS
2. Correct and check	29/7	ANO	29/7	AS				
3. Check estimates	30/7	ANO						
C. Trial data:-								
1. Plan and write trial data	2/8	ANO			4/9	ANO		
2. Punch and check trial data	9/8	ANO						
3. Prepare expected results	9/8	ANO						
4. Prepare control cards	9/8	ANO						
D. Initial compile etc:-								
1. Obtain clean compilation	23/8	ANO						
2. Dry-run program	23/8	ANO						
3. Check estimates	26/8	ANO						
E. Trials:-								
1. Run trials maintaining documentation	20/9	ANO						
2. Produce operating instructions	13/9	ANO						
3. Finalise documentation	20/9	ANO						

FORM 14/109/2(12.69)

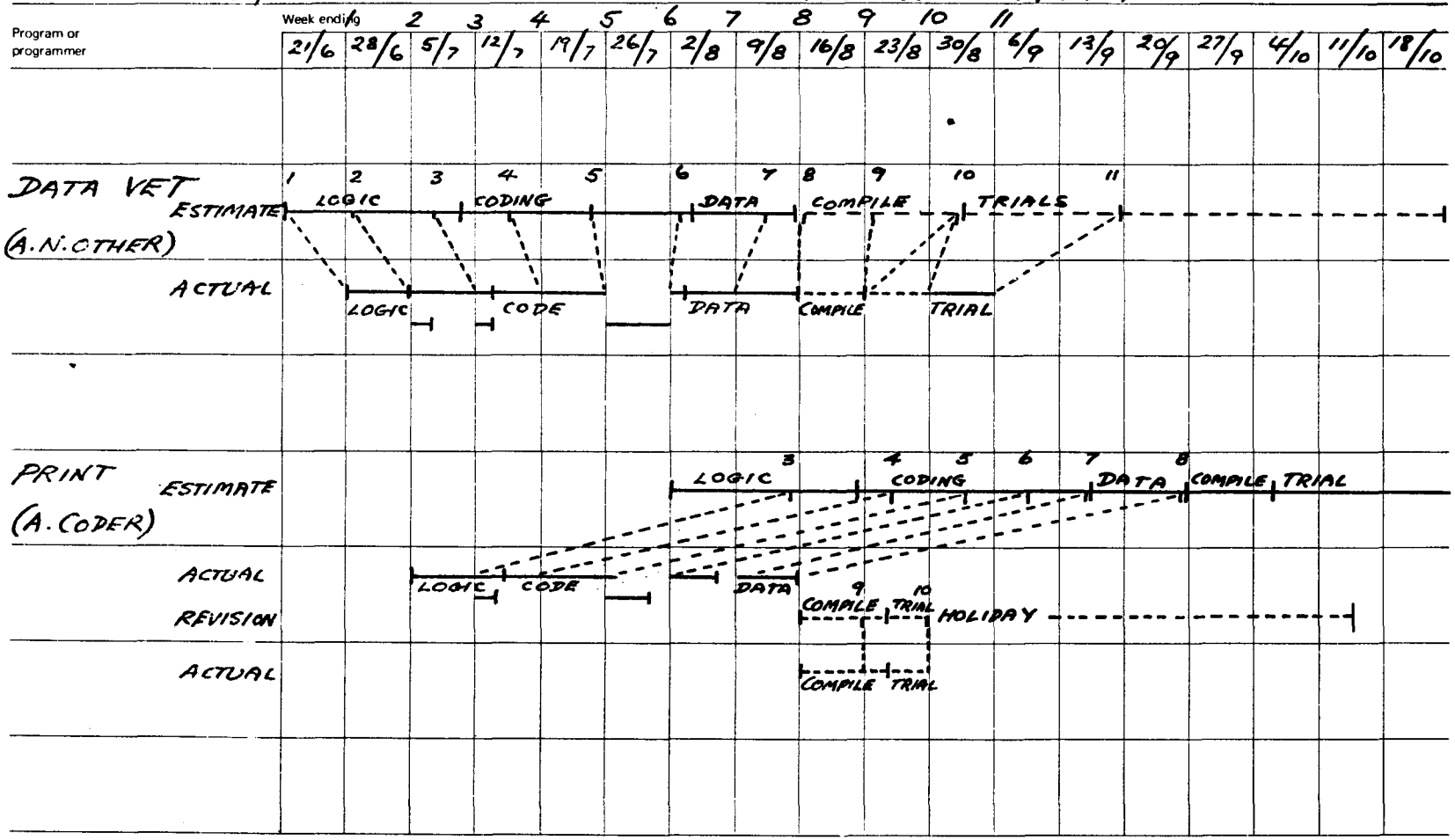
Example 2

Example 3

International Computers Limited — Data processing — **Programming schedule** — **ICL**

Prepared by A. SENIOR — User XYZ SERVICES LTD

Date 10.5.69 — Project PROJECT ANALYSIS



FORM 14/109/3(12.69)

International Computers Limited | Data processing | **Project progress report** | **ICL**

User XYZ SERVICES LTD Project PROJECT ANALYSIS Week ending 16.8.69.

To Name: L. JONES, J. SMITH, K. BROWN — Title — From: M. ROBINSON

Check list of entries

	1.	User organisation and staffing
	2.	Contacts and meetings with user (attach minutes)
	3.	Pre-programming: progress on job plan, system spec., program spec.
✓	4.	Program estimates and time-table
✓	5.	Programming: general and technical problems
✓	6.	Trials service
✓	7.	Staffing
	8.	Operational: progress and problems
	9.	Other comments

Entry no.

Chief programmer's comments

4 Print program re-estimated (lower than original). Card to tape and Print programs rescheduled to allow ½ time trials working and holiday. No change in overall end date.

5 Generally satisfactory. There is a system change which affects data vet and Update, but it is minor (about 4 days effort) and we plan to incorporate it during trials without causing delays.

6 Magnetic tape now available. Service good so far.

7 Staff: working well and morale improving now trials started.

M. Robinson

FORM 14/109/4(12.69)

Example 4

International Computers Limited Data processing **Program progress report** **ICL**

User XYZ SERVICES LTD Project PROJECT ANALYSIS Week ending 16.8.69

To: Name
L. JONES
J. SMITH
K. BROWN

Title

From:
M. ROBINSON

Program no. and description	Programmers	Progress and position: Programming	Linked/system trials
DATA VET A002A	A.N. OTHER	Current stage: Initial Compile % of current stage completed: 40% 3 DAYS REMAINING Comparison with time-table: ON SCHEDULE Comments: MAN-EFFORT AS EXPECTED. SYSTEMS CHANGE TO BE INCLUDED DURING TRIALS - NO EXPECTED DELAY	
		Current stage: % of current stage completed: Comparison with time-table: Comments:	
PRINT A008A		Current stage: Initial Compile % of current stage completed: 50% 1 DAY REMAINING Comparison with time-table: ON REVISED SCHEDULE Comments: RESTIMATED AT 200 INSTRUCTIONS LESS THAN ORIGINAL. RESCHEDULED FOR HOLIDAY AND ½ TIME WORKING. END DATE 2 WEEKS EARLIER. ABOUT 5 DAYS EXCESS EFFORT TO DATE.	
		Current stage: % of current stage completed: Comparison with time-table: Comments:	

FORM 14/109/5(12.69)

Example 5

Example 6

International Computers Limited — Data processing — **Program record** — **ICL**

Scheduled start date	17.6.69			Program no.	A002A
Scheduled end date	18.10.69			Program name	DATA VET

Est. number	Date	Programmer name	Man-effort (days) Logic	Coding	Trial data	Compile & dry-run	Trials	Total	Actual man-effort remaining	Expected man-effort remaining	Days ahead + behind −	Additional notes
1	10/5/69		11	15	6	5	15	52	52	52		[4 DAYS EXPECTED PER WEEK]
	21/6	A.N. OTHER							52	48	4−	UNSCHEDULED HOLIDAY
	28/6	"	4						48	44	4−	7 DAYS LOGIC LEFT
	5/7	A.SENIOR	4 1						42	40	2−	1 " " "
	12/7	A.N.OTHER A.SENIOR	1 1 ✓	3					38	36	2−	12 " CODING "
	19/7	A.N. OTHER		5					33	32	1−	7 " " "
	26/7	A. SENIOR		4					27	28	1+	1 " " "
	2/8	A.N. OTHER		1 ✓	3				22	24	2+	2 " TRIAL DATA "
	9/8	"			4 ✓				20	20	0	
	16/8	"				2			18	18	0	[2 DAYS EXPECTED PER WEEK] 3 DAYS COMPILE LEFT
	23/8	"				2			15	16	1+	
	30/8	"					4		10	14	4+	WORKING FULL-TIME
	6/9	"	½	1	½		2		8	12	4+	SYSTEMS CHANGE
	13/9	"					4		4	10	6+	
	20/9	"					2 ✓		0	8	8+	
		TOTAL	11½	14	7½	4	12	4				

FORM 14/109/6(12.69)

Example 7

International Computers Limited — Data processing — **Program record** — **ICL**

Scheduled start date 29/7/69 | 1/7/69 | | | | Program no. A008A

Scheduled end date 25/10/69 | 11/10/69 | | | | Program name PRINT

Est. number	Date	Programmer name	Man-effort (days) Logic	Coding	Trial data	Compile & dry-run	Trials	Total	Actual man-effort remaining	Expected man-effort remaining	Days ahead + behind −	Additional notes
1	10/5/69		11	15	6	5	15	52	52	52		[4 DAYS EXPECTED PER WEEK]
	5/7/69	A. CODER	4						45	52	7+	4 DAYS LOGIC LEFT
	12/7/69	" SENIOR	2 1✓	3					38	52	14+	12 " CODING "
	19/7/69	A. CODER		4					34	52	18+	8 " " "
	26/7/69	" SENIOR		1 3					30	52	22+	4 " " "
	2/8/69	A. CODER		3✓					26	48	2+	
	9/8/69	"			5✓				20	44	24+	
2	16/8/69		7	10	5	3	10	35	13	13	0	RESCHEDULED TO ALLOW HOLIDAY +2 DAYS PER WEEK
	16/8/69	A. CODER				3			11	11	0	1 DAYS COMPILE LEFT
	23/8/69	"				1✓	1		9	9	0	9 " TRIALS ✓ "
	6/9/69	"							9	9	0	2 WEEKS SCHEDULED HOLIDAY
	13/9/69	"					2½		7	7	0	
	20/9/69	"					1½		2	5	3+	
	27/9/69	"					2		2	3	1+	
	4/10/69	"					2		0	1	1+	
		TOTAL	7	14	5	4	9	39				

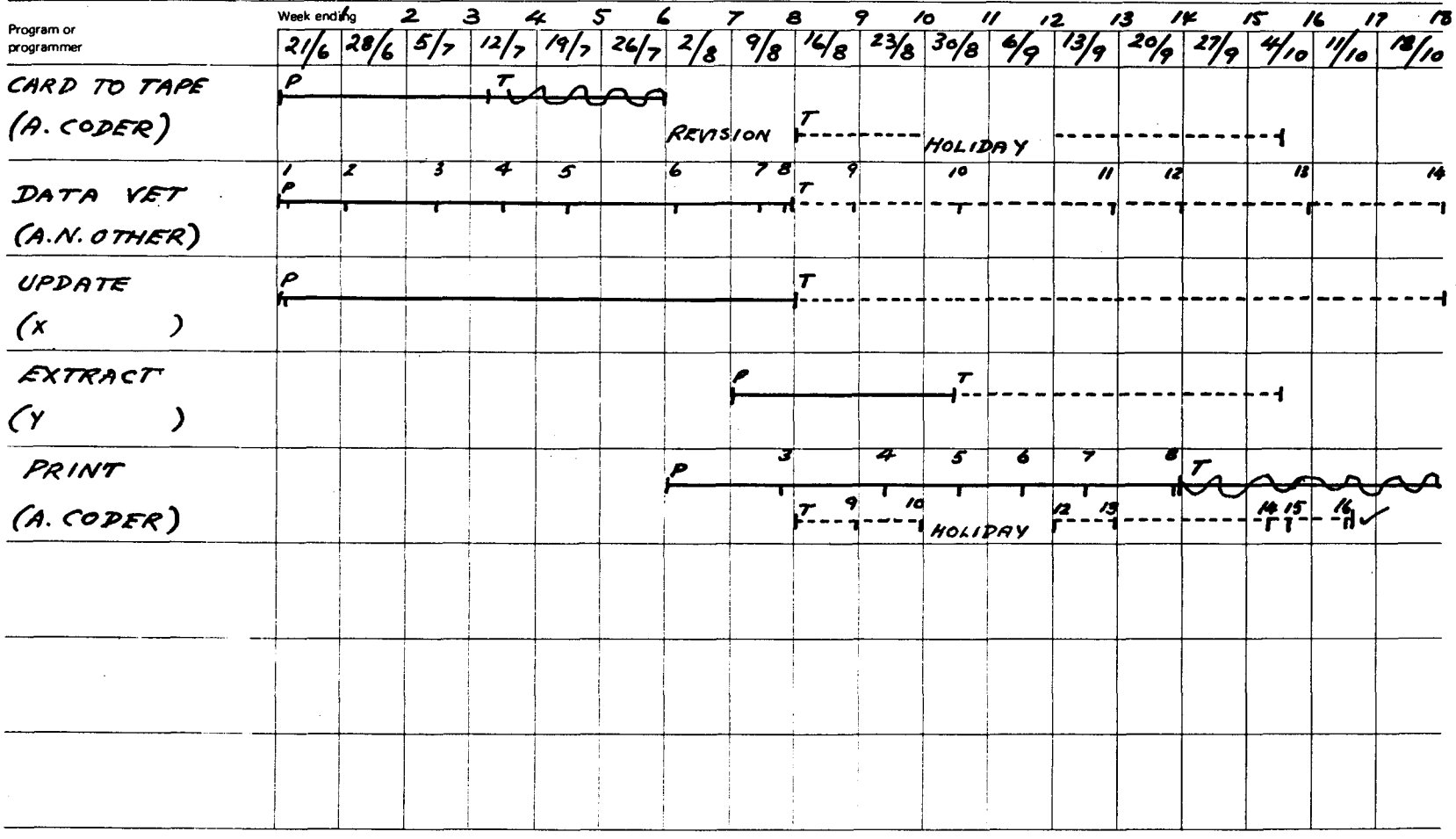
FORM 14/109/6(12.69)

Example 8

International Computers Limited Data processing **Programming schedule** **ICL**

Prepared by A. SENIOR User XYZ SERVICES LTD

Date 10.5.69 Project PROJECT ANALYSIS



FORM 14/109/3(12.69)

Example 9

International Computers Limited — Data processing — **Trial record** — **ICL**

Program no. A002A — Program name DATA VET

Estimate number	Date	Number of expected turn rounds per week	Data set up etc. Weekly No.	Data set up etc. Weekly Time	Data set up etc. Cumulative No.	Data set up etc. Cumulative Time	Initial Compile weekly No.	Initial Compile weekly Time	Program trials Compile weekly No.	Program trials Compile weekly Time	Program trials Trial weekly No.	Program trials Trial weekly Time	Total (*no: of turn-rounds) Weekly No.*	Total (*no: of turn-rounds) Weekly Time	Total (*no: of turn-rounds) Cumulative No.*	Total (*no: of turn-rounds) Cumulative Time	Notes
1	10/5/69	1/4				0.30									12	4.00	INITIAL COMPILES & PROGRAM TRIALS
	16/8						1	10					1	0.10	1	0.10	
	23/8						1	12					1	0.12	2	0.22	
	30/8		2	9	2	0.09			1	21	1	5	1	0.26	3	0.48	
	6/9		2	20	4	0.29			3	54	4	29	4	1.23	7	2.11	
	13/9		1	4	5	0.33			2	30	2	15	2	0.45	9	2.56	
	20/9				5	0.33			1	15	1	5	1	0.20	10	3.16	
		TOTAL			5	0.33	2	0.22	7	2.00	8	0.54			10	3.16	

FORM 14/109/7(12.69)

Example 10

International Computers Limited — Data processing — **Trial record** — **ICL**

Program no. A008A — Program name PRINT

Estimate number	Date	Number of expected turn rounds per week	Data set up etc. Weekly No.	Data set up etc. Weekly Time	Data set up etc. Cumulative No.	Data set up etc. Cumulative Time	Initial Compile weekly No.	Initial Compile weekly Time	Program trials Compile weekly No.	Program trials Compile weekly Time	Program trials Trial weekly No.	Program trials Trial weekly Time	Total (*no: of turn-rounds) Weekly No.*	Total (*no: of turn-rounds) Weekly Time	Total (*no: of turn-rounds) Cumulative No.*	Total (*no: of turn-rounds) Cumulative Time	Notes
1	10/5/69	2½				1.00									12	4.00	3 INITIAL COMPILES 9 PROGRAM TRIALS
2	12/8/69	1¼				1.00									9	3.00	3 INITIAL COMPILES 6 PROGRAM TRIALS
	16/8		1	17	1	0.17	3	38					3	0.38	3	0.38	
	23/8		2	10	3	0.27	1	17			1	7	2	0.24	5	1.02	
	6/9				3	0.27									5	1.02	
	13/9		4	39	7	1.06			3	35	3	13	3	0.48	8	1.50	
	20/9		5	45	12	1.51			4	55	3	20	4	1.15	12	3.05	
	27/9		1	7	13	1.58			4	61	5	43	6	1.44	17	4.49	
	4/10		3	19	16	2.17			2	22	4	26	4	0.48	21	5.37	
		TOTAL			16	2.17	4	0.55	13	2.53	16	1.49			21	5.37	

FORM 14/109/7(12.69)

International Computers Limited Data processing **Project history record** **ICL**

Project controller	J. SMITH	User	XYZ SERVICES LTD
Systems contact	K. BROWN	Project	PROJECT ANALYSIS
User contact	N. HARRIS (C.P.)	Suite no.	XYZ A0
Chief programmer	L. JONES		
Senior programmer	M. ROBINSON		

Computer	SYSTEM 4-50	Main run frequency	WEEKLY
Operating system	J	Main language	COBOL
Configuration	4 MT	Sub-languages	USERCODE
	2 RDS		
	CARD I/P LINE PRINTER		

Estimate number	No. instructions Low-level	No. instructions High-level	Specify Start date	Specify End date	Program Start date	Program End date	Linking Start date	Linking End date	System trial Start date	System trial End date	Trial time (hours) Program	Trial time (hours) Link
1	500	2400	START APRIL	END MAY	17/6/69	1/11/69	15/10/69	20/11/69	25/11/69	30/11/69	20	10
2	1500	2090	"	"	"	"	"	"	"	"	"	"
Actual	1670	2260	25/3/69	24/5/69	17/6/69	R/10/69	13/10/69	-	-	4/2/70	26¾	10¼

Actual

Staff category	Peak no. of staff	Average no. of staff used Specs.	Program	Link	System	Total	Man-weeks effort used Specs.	Program	Link	System	Total
Chief/senior programmers	2	2	1	1			11		16		27
Other programmers	4		4	2				29	8		37

Class of program	No. programs written	No. standard programs	Average run-time (MINUTES)	Program start point: Systems spec.	Draft program spec.	Agreed program spec.
Media conversion – input						
Media conversion with vetting	1		8			✓
Vetting without media conversion	1		7			✓
Media conversion – output						
Media conversion with editing	1		9			✓
Editing without media conversion	1		6			✓
Sorting/merging		3	4 EACH			
File updating	1		10			✓
File searching						
Other						
Total	5	3	52			

Please comment overleaf on any unusual programs and any special factors affecting implementation.

FORM 14/109/8(12.69)

Example 11

Example 12

International Computers Limited — Data processing — **Program history record** — **ICL**

Program number A0024 — Program name DATA VET

Estimate no.	Stage of * estimate	Date of estimate	Program language	No. of modules	No. instructions: Data	No. instructions: Procedure	Complex factor	Work content: units	Exp: factor	Total store	Time (days): Man effort	Time (days): Elapsed	Machine attempts: Number	Machine attempts: Time (hrs : mins)	Comments
1	P-S	10/5/69	COBOL	1		700	1	700	1	22,000	52	18wks	12	4.00	
2	P-C	31/7/69	COBOL	1	420	770	1	770	1	22,000	"	"	"	"	NO SCHEDULE CHANGE
Actual high level			COBOL	1	454	798	1	798	1	24,000	49	14wks	10	3.16	
Actual low level															

Actual store usage

Actual no: segments — Instructions + macros — File control

Actual no: overlays — Data + constants — Overheads

Actual implementation (man-days)	Logic and checking	Coding and checking	Trial data	Initial compile 2 dry-run	Trials	Total
	11½	14	7½	4	12	49

Actual trials (excluding link trials)	Initial compiles: Number	Initial compiles: Time	Compilations: Number	Compilations: Time	Trials: Number	Trials: Time
	2	0.22	7	2.00	8	0.54

Elements present in program:-

Vetting	✓	Media conversion	
Editing	✓	File updating	
Sorting		File searching	
Merging		Calculation	✓

*Abbreviation for stage of estimate:-

Pre-specification	Pre – S	Post-initial compile	P – I
Post-specification	P – S	Post-trials	P – T
Post-charting	P – Ch	Post-linking	P – L
Post-coding	P – C	Other revisions	R

FORM 14/109/9(12.69)

Example 12 (reverse of record)

Additional statistics

Vetting programs

Original input medium	MT	No. of input types requiring more than 1 block/card	NONE
No. of different input types	9	Average no. fields to be vetted on each input record	10
No. of output record types	11	Percentage of fields needing radix conversion before O/P	33%
No. of output files	1	Average data volume per run	1700 RECORDS AV. 91 CHARACTERS

Editing programs

Eventual output medium	PRINTER	Is two-up printing required	NO
No. of different output formats	10	Is pre-printed stationery required	NO
No. of input record types	9	Percentage of fields needing radix conversion before O/P	NONE
No. of input files	1	Average results volume per run	500 LINES

File updating and file searching programs
Give details of all magnetic tape/direct access and card/paper-tape files

Description of file	Input output both	Medium	Access method (discs)	Total file length	Average record length	No. of record types

Normal main file activity rate per run	%	If updating program, is a separate program used:-	Take-on	
			Amending	

Additional comments

(Please mention any points of special difficulty or interest, and any special factors affecting implementation)

Appendix 11 Trials procedures

PLANNING AND SUPERVISION OF TRIALS

There are two forms available which aid the planning and supervision of trials:

Testing schedule

Bar chart

Testing schedule

The testing schedule is used to draw up the basic trial data plan from which the programmer can develop a full set of trial data. It consists of a grid arrangement with one column for each significant circumstance and a row for each combination of circumstances to be tested.

The basic set of trial data should be drawn up by the programming supervisor. Basic combinations of circumstances and sequences of events should be defined. These will form the basis of, and guide the programmer in developing, a full detailed set of trial data.

The testing schedule contains the following entries:

Program number	Identity
Test unit	Identity
Prepared by	Programming supervisor's name
Date	Date form filled in
Trial data records	A reference to be assigned to each basic trial data record.
Conditions	Each column indicates a different circumstance. Each trial data record row indicates combinations of these circumstances.
Comments	A brief description of the objectives of the basic trial data defined.

Bar chart

The purpose of the bar chart is to aid the supervision of trials to the extent of following the actual progress compared with the estimated progress. A bar chart gives a pictorial record of the actual progress for the trials stages and test units compared with the planned strategy.

When initially drawing up a bar chart for a test unit the form is tentatively divided into as many sections as there are trials stages for that test unit. If there are a large number of entries, the bar chart can extend over several forms.

The bar marked Estimated number of effective trials (EER) is equally divided and numbered according to the estimated number of effective trials needed to complete the testing of that stage. In making this estimate, consideration should be given to the extent that the current stage has already been tested in the process of fully testing the previous stage.

The bar marked Estimated total number of trials (ETT) is then similarly completed. In the example on page 166, it has been assumed that 40% of trials are likely to be ineffective. This means that for every three trials in the EET bar, there are five corresponding trials in the ETT bar.

The final entries are the Estimated dates. These dates (usually week ending dates) correspond to the entries in the ETT bar according to the expected turn-round. In the example, the expected turn-round is one trial per week. There will only be one trial per turn-round for test units.

International Computers Limited

Data processing

Testing schedule

ICL

Program number	Test unit	Prepared by	Date

Trial data records	Conditions	Comments

FORM 14/108/1(1.70) FORM C14/108/1(1.70)

International Computers Limited Data processing **Bar chart** **ICL**

Program		Project	
Test unit		Prepared by	Date

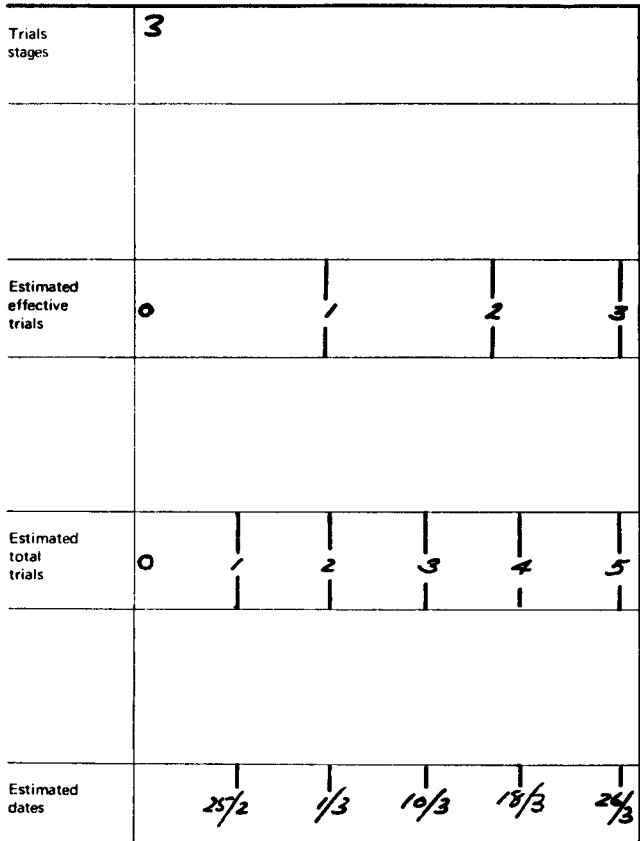
Trials stages	
Estimated effective trials	
Estimated total trials	
Estimated dates	

FORM 14/108/2(1.70) FORM C14/108/2(1.70)

International Computers Limited | Data processing | Bar chart | **ICL**

Program EXAMPLE Project PREPARING A BAR CHART

Test unit Prepared by Date

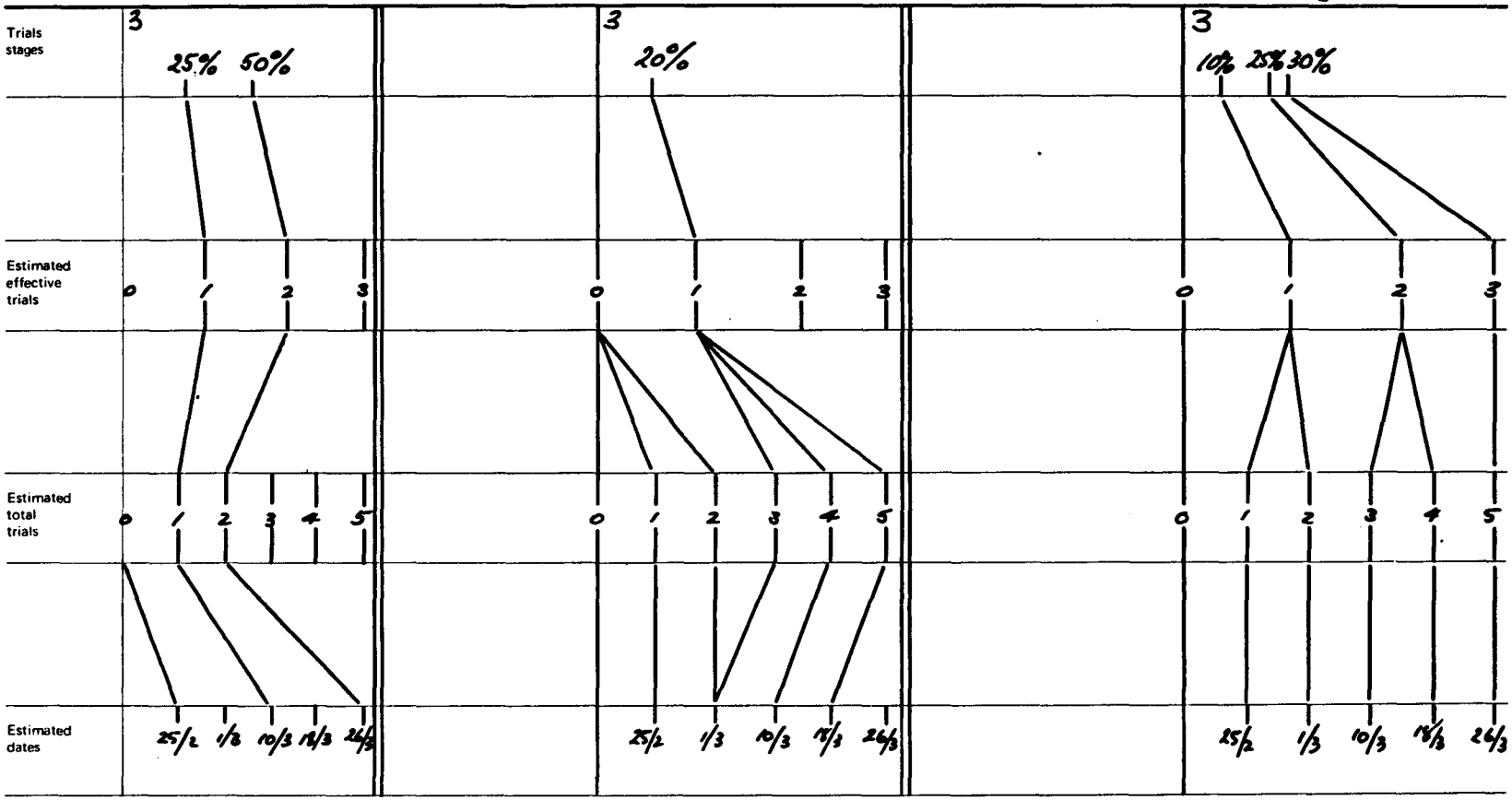


FORM C14/108/2(1.70)

International Computers Limited | Data processing | **Bar chart** | **ICL**

Program EXAMPLE | Project UPDATING AND INTERPRETING A BAR CHART.

Test unit A | Prepared by B | Date C



FORM C14/108/2(1.70)

International Computers Limited Data processing **Trials submission control** **ICL**

Job number		Submitted by	Date submitted	Description of jobs	Date returned
1	1	AB	1.1.69	Compile and Compose AAA1001.	10.1.69
	2	CD	3.1.69	Compose BBB1101 Test Units 3+4	
	3	EF	5.1.69	Trials of CCC1003 Test Units 1+2	
	4	GH	5.1.69	TDSU for DDD1004 Test Unit 1	
2	1	EF	10.1.69	Trial of CCC1003. Test Unit 2.	14.1.69
	2	AB	11.1.69	TDSU for AAA1001	
	3	MN	11.1.69	Compile, Compose, Trial XYZ1234. T.U.S. Stage 3.	

FORM C14/108/3(1.70)

Example of Trials submission control

UPDATING A BAR CHART

When updating a bar chart, preferably at least weekly, each trial for the test unit is considered independently. Firstly a line is drawn from the trial on the ETT to the actual date in the Estimated dates bar.

It has then to be decided whether the trial was effective or ineffective. If the trial was ineffective, a line is drawn from the trial in the ETT bar to the previous effective trial in the EET bar. This completes the entry for an ineffective trial. If the trail was effective, a line is drawn from the trial in the ETT bar to the next effective trial in the EET bar.

For the effective trial, a percentage estimate is made of how far it has progressed the testing of the trials stage. This is difficult to estimate and the programmer's opinion alone should not be used as it will probably tend to be optimistic. Instead, it should be based on the number of instructions proved or the number of trial data records proved. The percentage estimate is marked on the Trials stages bar and a line joining this and the effective trial in the EET bar is drawn. This completes the entries for an effective trial.

USING AND INTERPRETING BAR CHARTS

Bar charts for each test unit must be kept up-to-date. If the estimates are accurate then at the end of each trials stage, the lines drawn should form a single vertical line The divergence of these lines indicates the progress of testing. If discrepancies are abnormally large, then steps will have to be taken to correct the situation. Consider the three situations in the example on page 167.

If the Estimated total trials are not completed by the Estimated dates, then the programming supervisor should inform the programming management in order to improve the turnrounds.

If the Estimated effective trials fall too far behind the Estimated total trials then the causes should be investigated. If the programmer is at fault, then stricter checking procedures should be enforced. If the reasons are machine, operator or software faults then the programming management should be informed.

If the progress of a trials stage is slower than estimated against the effective trials then either the program should be dry-run again or if similar situations are occurring on the other test units, then perhaps all the estimates for that stage or even all the stages may have to be revised.

When a revision has to be made, a new bar chart should be drawn up for the remainder of the testing. The date and reason for the revision should be noted on the bar chart. The revised bar chart may commence on a previous partially completed form or on a new form.

CONTROL OF TRIALS IMPLEMENTATION

The day by day control of trials implementation is essential for the smooth running of a project. If individuals are left to their own devices, then chaos will result. A programmer's natural optimism will mean that both checking and co-ordination between each individual's submission will be insufficient. There are four forms available which will aid the programming supervisor in controlling trials, whether by his own use or by suitable delegation to a trials controller. These are:

Trials submission control

Turn-round

Tape/disc catalogue

Data prep submission

Trials submission control

The Trials submission control form provides an interface between the programmer and the machine. It mainly contains information on the turn-round of the programmer's work.

The main control here is that the programmer cannot submit directly to the machine. Before accepting a programmer's submission, the controller should check that the programmer has created his submission in the correct manner. This will include ensuring that the analysis has been checked to be full and complete; that the programming supervisor has approved the resubmission; that all the submission details have been checked.

The form also provides a check on the work in hand in case any submission is misplaced or overlooked.

Once a submission is accepted it can then be considered as part of the total work submitted.

International Computers Limited Data processing Turn-round **ICL**

Job number	Date submitted	Date returned	Machine time	Result	Comments
1	6.1.69	10.1.69	2hrs 15 mins		
4			30 mins	E	
1			30 mins	E	
3			30 mins	E	
1			15 mins	I	T.U.2. Op. mistyped reply.
2			30 mins	E	
2	11.1.69	14.1.69	27 mins		
2			10 mins	I	Card reader misreading
1			12 mins	E	
3			5 mins	I	Complication due to mispunched parameter cards - Programmer.

FORM C14/108/4(1.70)

Example of Turn-round

The Trials submission control form contains the following entries:

Job number	A unique reference number for each individual job submitted
Submitted by	Programmer's name
Date submitted	Date job is submitted by the programmer
Description	Brief details of the function of each job
Date returned	Date job is returned to the programmer

Turn-round

The Turn-round form provides an interface between the controller and the machine. It mainly provides detailed information on the total turn-round of work. The controller oganizes the total work accepted for submission as defined for the installation, and records this on the form as appropriate.

The exact overall use of machine time can be readily deduced from the information recorded on the form. Tendencies in turn-round times can be analyzed and may result in steps being taken to improve the situation or possibly a revision of estimates. The frequent occurrence of errors which produce ineffective results may be discovered and remedial action may be taken especially if they were caused by the programmer.

The form can also provide information for costing, timing, efficiency, and other required aspects of testing.

The Turn-round form contains the following entries:

Job number	The job numbers in the order they are to be run within a batch of jobs
Date submitted	The date the batch of jobs is submitted
Date returned	The date the results of the batch are received
Machine time	Amount of machine time used
Result	E for effective I for ineffective
Comments	Relevant details and comments on results. Further details are held on the Trials log/analysis (described at the end of this appendix).

Tape/disc catalogue

The Tape/disc catalogue form provides a basis for a method of controlling the use and contents of the pool of tapes and discs. There is one form for each tape or disc and the supervisor keeps the library of forms and controls their updating. He should control the allocation and use of the tapes or discs according to their availability and any special priorities there may be.

The Tape/disc catalogue form contains the following entries:

Project	Name of project(s) to which the tape/disc is assigned
Serial number	Reference number of tape/disc
Location	Location of tape/disc
Owner	Person responsible for the tape/disc or POOL if unassigned
Date	Date of any relevant change
Type	Description, for example short/long work or data tape
Contents	Details of files, standard or user labels of any data held, including position if on disc
Run generation number	Relevant value
Condition	History of tape/disc, (parity, scratches, tape chopping etc.)

Data prep submission

The Data prep submission form provides a means of controlling the flow of data preparation work. This will be useful if there are large numbers of submissions. Work can perhaps be distributed to different data preparation

International Computers Limited Data processing **Tape/disc catalogue** **ICL**

Project

Serial number AAA111

Location	Owner	Date	Type	Contents – files/labels/data	Run/ generation no.	Condition
H.H.	A.B.	1.1.69	Long Data Tape	Standard Labels. Test data for AAA1001.	1	O.K.
				3 files for stages 2, 3 and 4.		
		20.2.69		Added file for stage 5.	2	
		26.2.69				Parity. Small
						mark on tape.
						cleaned.
C.H.		3.3.69				
	POOL	6.4.69				
C.O.	CD	8.4.69	"	Standard Labels. Cheque file for BBB1002	1	

FORM C14/108/5(1.70)

Example of Data prep submission

International Computers Limited | Data processing | **Data prep submission** | **ICL**

Job number	Programmer	Description		No. of cards/chars. Estimate	Actual	D.P. centre	Date submitted	Date received	Time	Remarks
AAA1001	A.B.	Program	PC	2,000	2,184	H.H.	3.1.69	7.1.69	55mins	Several errors
"	"	Parameters	PC	100	89	"	"	"	10mins	O.K.
"	"	Parameters	PT	1,000	1,304	C.H.	"	6.1.69	25mins	O.K.
BBB1002	CD	Amendment	PC	150	165	H.H.	"	7.1.69	6mins	Several errors
AAA1001	AB	Test Data (St. 2)	PC	50	61	H.H	9.1.69	11.1.69	5mins	O.K.
AAA1002	"	Parameters	PT	50	49	C.H.	"	12.1.69	5mins	O.K.

FORM C14/108/6(1.70)

centres as required. The form provides the necessary information readily available in case of any queries. Tendencies in turn-round times can be analyzed. This will enable estimates to be made of expected delays which will enable other work such as dry-running and checking, to be scheduled more accurately.

The Data prep submission form contains the following entries:

Job number	Reference number of work submitted
Programmer	Name of programmer who submitted work
Description	Relevant details such as paper tape, verify, hard copy, parameters

Number of cards/characters:

Estimate	Number estimated by programmer
Actual	Number given by data preparation department
D P Centre	Punching installation used
Date submitted	Date the work is submitted
Date received	Date the completed work is received
Time	Time taken to process work
Remarks	Details of mispunching etc.

PROGRAMMERS TESTING PROCEDURES

The summaries below provide a check list of the programmer's functions involved in testing. The functions are not necessarily in the exact chronological order of execution. Specimen testing schedule and Trials log/analysis forms with descriptions of their use appear at the end of this appendix.

Trial data

1 PLAN Full sets of trial data should be planned to test all conditions on the basis outlined by the supervisor on the testing schedule form(s). A testing schedule form should also be completed by the programmer (see following instructions)

2 APPROVAL The supervisor should approve the objectives of the full sets of trial data to avoid unnecessary errors and amendments later on

3 DETAILS The exact data record details should be prepared for all conditions planned. The associated expected results must also be prepared in detail

4 PREPARATION The trial data should be submitted for punching as applicable

5 SET-UP The trial data is set up (for example, a main file on magnetic tape) if required

6 CHECK The trial data files set up and the associated expected result should be checked to be correct. Also a check should be made to ensure that the trial data meets the requirements of the objectives

7 DOCUMENTATION All relevant documentation must be kept up to date

Initial procedures

1 DIAGNOSTICS Suitable basic diagnostic facilities may be incorporated in the program to increase the value of trials and to aid analysis

2 APPROVAL The supervisor must approve the proposed use of diagnostics to avoid unnecessary wastage of machine time

3 PREPARATION The program should be submitted for punching as applicable

4 CHECK The results of the preparation should be checked to be correct

5 INITIAL COMPILATIONS The program should be submitted for compilation via the relevant channels. All errors must be fully analyzed and corrected. Full details must be recorded on the Trial log/analysis form

6 CHECK A final standards check and a final logical check should be arranged, using the initial compilation listings

7 DRY-RUNNING A thorough dry-running using an initial compilation listing and trial data, will increase efficiency of trials and will validate trial data. A second person is required for assistance

8 DOCUMENTATION All relevant documentation must be kept up to date.

Trials

1 DOCUMENTATION Full details of trials should be recorded on the Trials log/analysis form for checking, change of staff and maintenance purposes, (see following instructions). All relevant documentation must be kept up to date

2 PREPARATION Parameters and operating instructions for the trial should be prepared, the relevant submission details should be recorded

3 CHECK A check of the submission details should be arranged

4 SUBMISSION The trial is submitted via the relevant channels

5 ANALYSIS Failures must be diagnosed; all errors must be analyzed and full details must be recorded

6 CORRECTIONS Amendments must be made for all errors, and full details must be recorded

7 CHECK A check of the analysis and corrections should be arranged

8 REPEAT The procedure should be repeated for each trial

Testing schedule

This testing schedule form provides both an aid to devising, and a means of recording, trial data. It consists of a grid arrangement with one column for each significant circumstance and a row for each combination which is to be tested. The testing schedule may be a modification of the appropriate decision table if decision tables were used to define the job.

A schedule of the basic trial data plan is drawn up by the programming supervisor.

A separate detailed schedule is drawn up by the programmer

The testing schedule contains the following entries:

Program number	Identity
Test unit	Identity
Prepared by	Name of person designing the trial data
Date	Date form filled in
Trial data records	This enables a reference to be assigned to each trial data record
Conditions	Each column indicates a different circumstance. Each trial data record row indicates combination of these circumstances.
Comments	This can be used to give a brief description of the objectives of trial data sets and/or each individual trial data record.

Trials log/analysis

The Trials log/analysis form provides the following advantages if the entries are meticulous:

1 A comprehensive resumé of the testing progress to-date

2 A valuable record if the program is to be continued by another programmer

3 A useful aid for checking procedures

4 A source of information for statistical purposes

The trials log/analysis form contains the following entries:

Program number	Identity
Program name	Identity
Test unit	Unit of coding being tested

Example of Testing schedule

International Computers Limited | Data processing | **Testing schedule** | **ICL**

Program number (EXAMPLES) Test unit Prepared by A.N. OTHER Date

Trial data records	Conditions																												Comments
	COMPANY	DEPARTMENT	CODE : 618	GRADE < 10	GRADE > 20	TAX CODE < 500	500 ≤ TAXCODE < 600			AMENDMENT	INSERTION	DELETION	CODE	IDENTITY															
1	X	A	=	✓		✓				✓			1																Simple straightforward tests
2	X	B	<	✓		✓					✓		1	✓															
3	X	A	=	✓			✓					✓	1																
4	X	B	>	✓			✓			✓			2																
5	X	A	=		✓	✓					✓		2	✓															
6	X	B	<		✓	✓						✓	2	.															
7	X	A	=		✓		✓																						
8	X	B	>		✓		✓																						
9, 10	Y	A	≠	✓		✓																							Simple error path tests
11, 12	Y	B	≠	✓		✓																							

FORM C14/108/1(1.70)

Operating system	Identity
Programmer	Identity

SUBMISSION

Trial number	Each trial is given a number in ascending numeric order starting at one.
Event	Type of run, (trial, compile, etc.)
Checked by	Signature of person who checks the submission.
Date submitted	Date of submission to the trials controller (or to the computer).
Date received	Date received from the trials controller (or from the computer).

RESULTS/RESUBMISSION

Result	E for effective I for ineffective
Approximate computer time	Amount of machine time used.
Details, analysis, corrections, comments, diagnostics	Full details of errors for all events. This is used mainly to record full details of results, analysis, corrections and comments on trials.
Checked by	Signature of programming supervisor or person who checks that analysis is correct.

International Computers Limited | Data processing | **Trials log/analysis** | **ICL**

Program number ABC 1234 | Program name EXAMPLE | Programmer A.N. OTHER

Test unit 3 | Operating system 5J

Submission					Results/Resubmission							
Trial no.	Event	Checked by	Date submtd.	Date recd.	Result	Approx. comp. time	Details. Diagnosis. Analysis. Corrections. Comments.					Checked by
1	Trial	A.B	1.1.70	3.1.70	E	5mins	Address error early on because second base register not set up correctly - displacement 4095 instead of 4096. Amended to add 1 to contents of 2nd base register. 0%					Z.Y.
2	Trial	A.B.	4.1.70	5.1.70	I	7mins	Would not read input as MT FCP module included instead of PT FCP module - Re-compose. 0%					Z.Y.
3	Trial	A.B.	5.1.70	8.1.70	E	5mins	Fully tested the check-digit routine successfully. Looped in small edit routine. Counter changed from 10 to 9. Should get to end next time. 20%					Z.Y.
4	Trial	M.N.	10.1.70	11.1.70	E	8mins	Spurious error messages but struggled to END. 3 edit patterns slightly wrong due to miscalculation. Branch wrong way so code 6 is treated same as code 4. 75%					Z.Y.
5	Trial	A.B	12.1.70	13.1.70	E	6mins	Main bulk of program correct. Reconciliation total error due to picking up values from wrong section of tables. Should be perfect next trial. 90%					Z.Y.
6	Trial	A.B.	14.1.70	16.1.70	E	7mins	PERFECT 100%					Z.Y.

FORM C14/108/7(1.70)

Appendix 12 Flowcharting standards

This appendix describes the standards for flowcharting and for cross-referencing coding to flowcharts. The purpose of any flowchart is to make clear what the program does without the obscurity of too much detail, and the main objectives of the standards described in the following sections are:

1 To make the flowcharting phase more effective by introducing various levels of charts which correspond to the level of processing to be solved

2 To enable flowcharts to be kept up to date more easily

3 To introduce a logical system of cross-referencing between coding and flowcharts

4 To make a flowchart drawn by any programmer intelligible to any other programmer, by using standard techniques

TYPES OF FLOWCHART

In most projects there are at least three different types of flowchart, these being:

The system flowchart

This is the highest level flowchart. It shows the relationship between the user departments and the computer system.

The suite organization flowchart

This is developed from the system flowchart and shows the organization of related programs and the flow of data and results to and from each program or associated manual process.

The program flowchart

Program flowcharts are drawn for each program in the suite. They consist of a set of multi-level flowcharts each of which is a complete flowchart. These are the subject of this document and are considered in greater detail.

THE PROGRAM FLOWCHARTS

The program flowcharts consist of two parts; an outline flowchart, which shows the overall logic of the program, and detailed flowcharts which are expansions of the outline chart.

Outline flowcharts

The outline flowchart is the highest level of the program flowchart. The level of detail is such that the flowchart is applicable to any computer which has similar peripheral and storage facilities.

The aim is to define the main logic of the program, in particular input and output, in such a way as to divide the program into logical stages each of which is a self-contained entity. All attention should be concentrated on establishing the basic logic of the program.

Each logical stage is represented by a symbol on the flowchart, and entries from other stages must be to the start of a stage, so that each stage is preserved as a self-contained unit.

Detailed flowcharts

The detailed flowcharts are based on the outline flowchart.

The aim is to include all the detailed requirements of the specification, maintaining the logic already established in the outline flowchart. The resulting flowcharts should contain detailed computer solutions for the logical stages on the outline flowchart, since these charts form the basis for coding the program.

A separate chart is drawn for each box on the outline flowchart that requires more detail, and these charts form the second level of program flowcharts. Further charts may be drawn to expand the detail of boxes on the second level charts, and these charts form the third level of program flowcharts. This process of expanding boxes to a lower level of chart is continued until at the lowest level the procedures can readily be expressed at coding level.

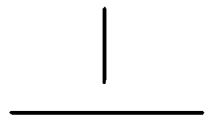
Since each chart represents the expansion of one box of a higher level chart, entries to a chart from other charts will always be to the first box. Flow lines would otherwise be implied as entering into the middle of the symbol on the higher level chart.

FLOWCHART SYMBOLS

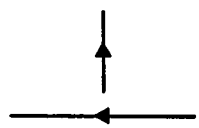
The following symbols, which are in accordance with the European Computer Manufacturers Association (ECMA) standards for program flowcharts, are to be used when flowcharting. Symbol size may be varied but the shape must be easily recognizable. These symbols are all contained on the ICL template.

FLOW LINES

A flow line shows the transfer of control from one operation to another. The standard direction of flow is from top to bottom, left to right.

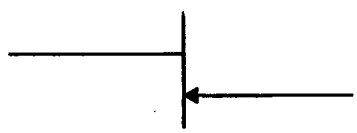


Arrows indicating the flow must be used when the direction of flow is not standard, or whenever increased clarity will result.



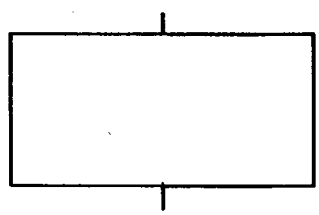
Flow lines which are unrelated may cross, but must never cross other unrelated symbols.

Flow lines may be connected, but only one entry point is allowed to a symbol therefore flow lines must join before entry.



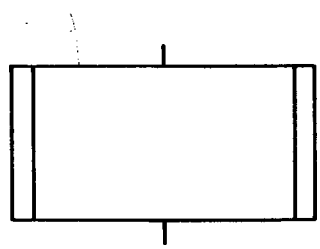
PROCESS

This symbol represents any kind of processing function or operation for which no particular symbol is provided.



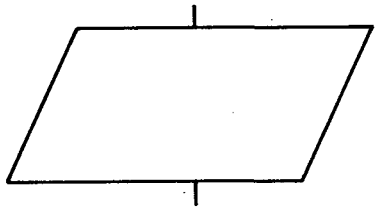
PRE-DEFINED PROCESS

This symbol represents a named process consisting of one or more operations or program steps that are specified elsewhere, for example, a subroutine.



INPUT/OUTPUT

This symbol represents an input/output function, that is where information is made available for processing, or processed information is recorded.

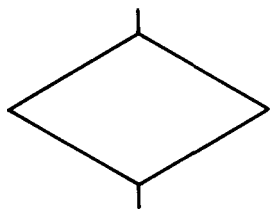


PREPARATION

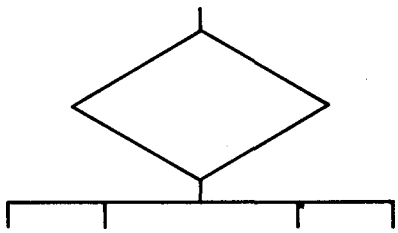
This symbol represents an instruction or group of instructions which indirectly affect the program flow, for example set a switch, modify, initialize.

DECISION

This symbol represents a decision or switching operation. It has one entry and a number of exits. Each result is stated alongside its appropriate exit.

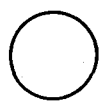


If more than three exits are necessary this symbol is used.



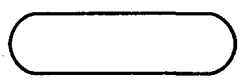
CONNECTOR

This symbol represents an exit to, or an entry from, another part of the flowchart. It is used to indicate a transfer of control which cannot be conveniently shown by a flow line (for example, where the destination is on another page, to avoid crossing flow lines or to avoid very long flow lines.) The symbol must contain a reference to establish the path.



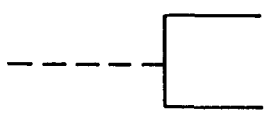
TERMINAL

This symbol represents a terminal point in a flowchart, for example start, stop, halt, interrupt.



COMMENT

This symbol is used to add descriptive comments or explanatory notes to a flowchart.



CONVENTIONS

Paper size and direction of flow

Flowcharts are drawn on A4 or A3 size paper, subject to the limitation that no page may contain more than 25 boxes. The paper is held vertically and the general direction of flow is from top to bottom, left to right.

Titling

Flowcharts must be clearly titled, dated, labelled and referenced, including the author's name. Page numbers must also be entered.

Text

All text associated with a symbol is written in block capitals, starting if possible with a verb. Where possible the text is written within the symbol, but if this is not practical the text may be written in a 'comment' symbol.

Subroutines

Subroutine entry points are annotated to indicate the function of the routine and the identities of all pages from which the routine is entered. The conditions required for entry to the subroutine should also be stated at this point.

Where a symbol represents a Library subroutine, the Library subroutine name should be entered in the box as well as a description of its function.

Modules and segments

The comment symbol should be used on the outline flowchart to show which boxes are to be grouped together to form modules or segments.

FLOWCHART REFERENCES

Referencing method

Each box on a flowchart is given a reference letter or letters. This reference may be written inside or outside the box at the top right. Whichever method is adopted, it must be used consistently throughout the charts. Any box which is expanded to a lower level chart must also contain an asterisk.

The boxes on the outline flowchart are referenced by a single letter in the range A to Y. The letter Z is reserved for subroutines, which are referenced in the range ZA to ZY, ZZA to ZZY, etc.

At the next level the reference of the first box on the chart and the chart reference are the same as that of the box which is being expanded. The references of other boxes on the chart are formed by suffixing the chart reference with the letters A to Y.

At subsequent levels this concept is continued as far as is required, following the rule that the first box on the chart and the chart reference is the reference of the box being expanded and further boxes within the chart are referenced with the chart reference and a suffix in the range A to Y.

Restrictions

The first box on any detailed flowchart must not be expanded. This avoids any two charts having the same identity.

Note: An alternative system is to reference the first box on a chart by adding the letter 'A' to the reference of the box being expanded but this system is not recommended since the inter-chart referencing system does not then follow the outline flowchart referencing.

The limitation on the letters that may be used implies a maximum of 25 boxes per chart. However, it is recommended that a limit of 20 boxes per chart is imposed, as this allows alterations to be made without extensive relabelling. In any case this is probably more than can be conveniently drawn on A4 paper.

Connectors

Exit connectors contain the logical reference of the destination symbol, and entry connectors contain the logical reference of the source symbol. Where several entries are made to a box the comment symbol may be used to show the incoming references. Wherever possible entries from and exits to other segments should include a segment reference.

There are two alternative ways of showing the visual connections via page numbers.

1 Connectors between charts should contain the appropriate page number, and boxes which are expanded should contain the page number of the chart containing the expansion. This method has the disadvantage that these page numbers will have to be entered after all the charts have been completed since when the charts are being drawn the page numbers have not necessarily been established

2 A cross-reference list of chart references and page numbers should preferably show the chart references in alphabetic sequence. This method has the disadvantage that the actual charts do not show page number connections but the list is easy to construct

CROSS-REFERENCING CODING TO FLOWCHARTS

The system adopted in cross-referencing will vary slightly according to the language being used. Flowcharts and coding must provide the authoritative working and maintenance documentation for the program at all stages of development. The structure and detail of both must correspond as closely as possible throughout. Since the flowchart is the explanatory detail of the task which the coding is designed to do it is desirable that the flowchart references form some part of the program labels. However where alpha labels are not permitted by a language the labels used in coding must be written above the appropriate flowchart box.

Program labels

The first line of coding to implement a box on a flowchart has a label of the form 'L' followed by the flowchart box reference. For example the coding label for the first instruction of box BC will be LBC.

Labels required for transfers of control within a flowchart box use the box label plus a numeric extension in the range 01-99. For example, the first transfer of control within box BC will be LBC01.

Subroutine labels

Since subroutine chart references all start with Z, labels for subroutines are a copy of the flowchart box reference, followed by the numeric extension as for program labels.

Label extension

In COBOL a meaningful paragraph name should be added to the standard label. Since the standard label can vary in number of characters the two parts should be separated by a hyphen.

Example

LC-MATCHDATA

ZDAB-STERLCONVERT

In languages where there are insufficient characters allowed for an extension, annotation should be included in the coding so that all major entry points are meaningfully described.

PROGRAM FLOWCHART EXAMPLE

Following is an example of a flowchart using the methods described in this appendix.

PROGRAM FLOWCHARTS

Name of job DEMON

Program identity : DEMO 1

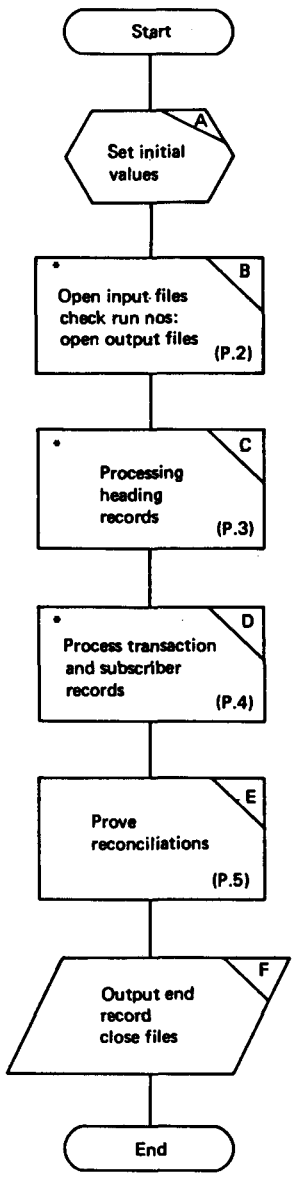
Chart reference	Page no.	Chart reference	Page no.	Chart reference	Page no.	Chart reference	Page no.	Chart reference	Page no.
Outline	1								
B	2								
C	3								
D	4	DA	6						
		DB	7						
		DD	8	DDA	9				
E	5								
ZA	10								
ZB	11								
ZC	12								
ZD	13								

ICL Data processing **Programming stationery**

Use Flowchart Outline

Title DEMON PROJECT Program/segment ref. DEMO 1

Prepared by: A.N.OTHER Date: 12/3/70 Checked by: J. BULL Date: 18/3/70

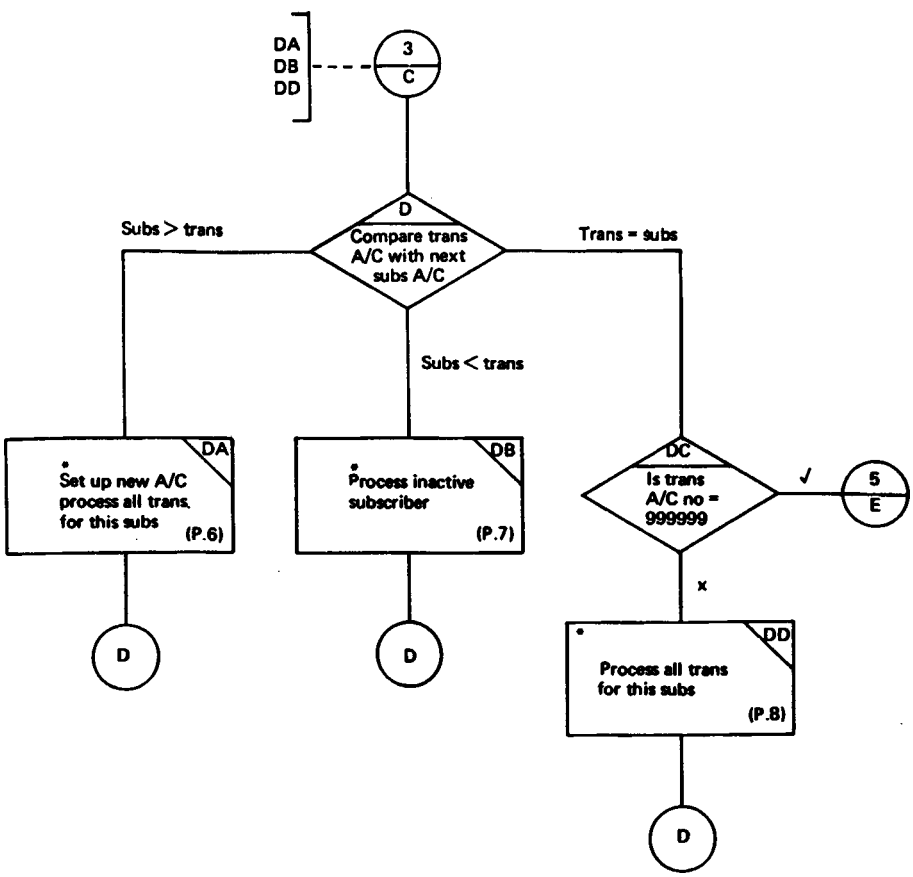


ICL Data processing **Programming stationery**

Use Flowchart D

Title DEMON PROJECT Program/segment ref. DEMO 1

Prepared by: A.N. OTHER Date: 12/3/70 Checked by: J. BULL Date: 18/3/70

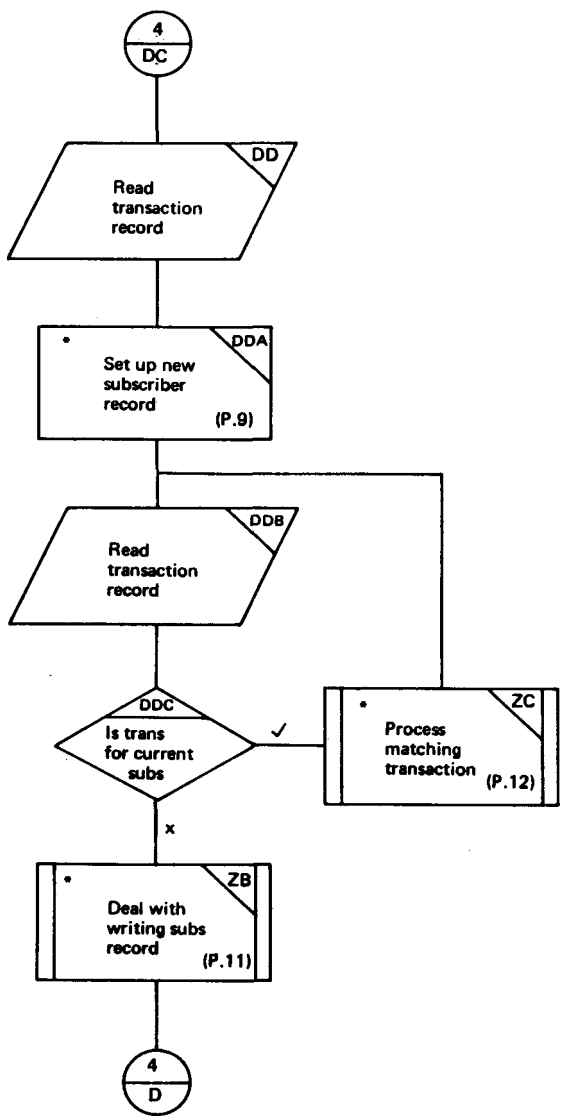


ICL Data processing **Programming stationery**

Use Flowchart DD

Title DEMON PROJECT Program/segment ref. DEMO

Prepared by: A.N. OTHER Date: 12/3/70 Checked by: J. BULL Date: 18/3/70

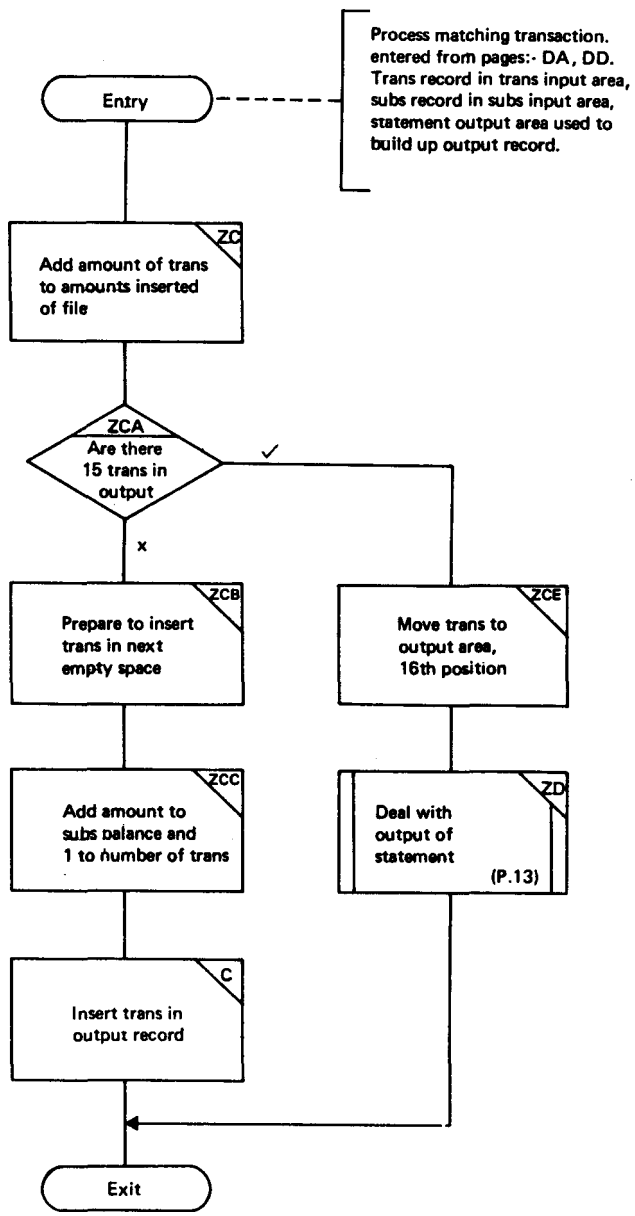


ICL Data processing **Programming stationery**

Use Flowchart DC

Title DEMON PROJECT Program/segment ref. DEMO.

Prepared by: A.N.OTHER Date: 12/3/70 Checked by: J. BULL Date: 18/3/70



Page 12 of 13

FORM C5/25(7.69)

Appendix 13 PLAN standards and techniques

This appendix covers standards for the writing of data names, labels, subroutine names, listing control, switches, monitor, subroutine linkage, and multi-segment programming. These standards are intended to assist the programmer in laying out his program so that it may be understood by another programmer more easily than if no convention had been followed. Clarity of both layout and intention is the aim. Wherever possible, the conventions for PLAN match those of other ICL languages.

WRITING STANDARDS

Writing

Characters should be clearly legible. At least every other line on the coding sheet should be left blank, to enable amendments to be easily inserted: even if amendments are not anticipated, the blanks greatly increase the legibility of the coding. It may be also useful to have about five blank lines at the end of each sheet.

Layout

It is essential to the maintenance of a program that it be laid out in a neat and standard manner.

#, #CUE and #PAGE

These directives should be used to mark logical groups of coding, and to space the listing. The programmer should remember, however, that macro expansions will of themselves create spaces, though not necessarily at logical breaks in the program. Thus a blank comment line (#) need only be used whenever a logical break occurs without a nearby macro space, and to separate groups of constants and work areas.

All routines and subroutines should begin with #PAGE, unless they are very small, when two blank comment lines should be used. In addition, the #PAGE directive should be used to separate the major logical sections of routines.

The #CUE directive makes a very useful logical separator. Its use should not be restricted to entry points from other segments as it can with advantage be used thoughout the coding. The cuename will be included in the consolidated list, simplifying the analysis of error dumps to some extent. Subroutines, in any case, should be CUEd.

Section order

Each section of a program should start on a new page, and be arranged as far as possible in the following order:

1 All COMMON areas

2 Variables (Lower)

3 Variables (Upper)

4 Presets (Lower)

5 Presets (Upper)

6 Subroutines

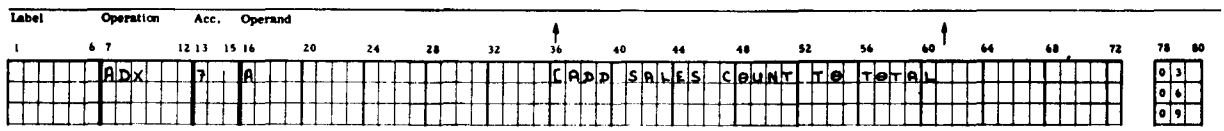
7 Program

In order to ease maintenance, within these sections, all labels and data names should as far as possible be in alphabetical order. However it is recommended that all file areas (see section *Data names* for a description of these) appear at the head of their relevant section. Thus file buffer areas will normally head section 2, file control areas section 4. Variables are placed before presets so that variable addresses (for example, buffer areas) are defined for use in presets (for example, file control areas). Note that contrary to the practice in most languages, subroutines precede the main program. This has the effect of reducing the size of the branch-ahead table.

Comments

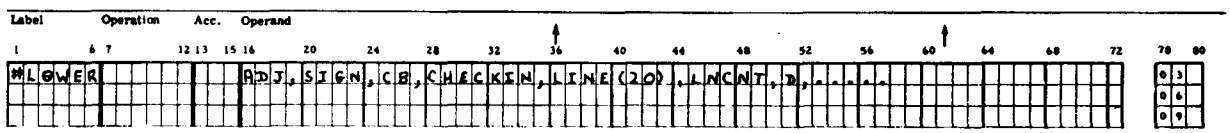
The importance of comments in low-level programs cannot be over-stressed. Not only do they greatly assist others to understand the program (as well as assisting the author who may forget his intentions) but also relate the coding to the flowchart, and constitute an aid to clear thinking. It is not possible to give strict rules for the number of comments per instruction, since this depends very much on the complexity of the coding, but certainly the aim is to produce a reasonably self-explanatory program. It is far better to err on the side of too much narrative than too little although the narrative must also be meaningful.

Example



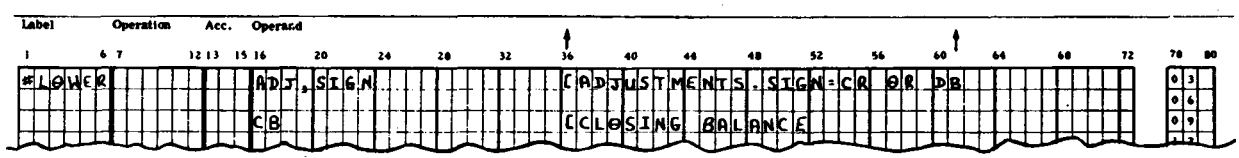
A much higher proportion of comments should be used in declarations of constants, work areas, etc., to describe the intended use of the areas. Clearly, lists of variable declarations are undesirable.

Example



Such lists must be written on many more lines.

Example



Comments should also be used as headings for groups of constants, file areas, etc. Branch instructions, particularly branches to other segments, should be well explained. All these points are illustrated in the examples at the end of this appendix.

NAMES

This section describes a method of naming which ensures as far as possible the uniqueness of names, enables the programmer to trace data references easily, and so order his data such that a good program layout is achieved. An important consideration in the method is that as few characters as possible should provide a maximum of clarity; this is necessary for PLAN 1, and helpful in PLAN 3 where compiler table sizes can be kept small without affecting the power of the language.

Data names

The method of naming data is based on the concept of a file. Wherever possible, all variables and presets associated with the same file should be grouped together into a file area. All the names in a file area begin with the two character peripheral code, which may be:

CP Card punch

CR Card reader

CT Cassette tape

ED Exchangeable disc store

FD Fixed disc store

LP Line printer

MC Magnetic card file

MT Magnetic tape

TE Twin exchangeable disc store

TP Paper tape punch

TR Paper tape reader

TW Typewriter

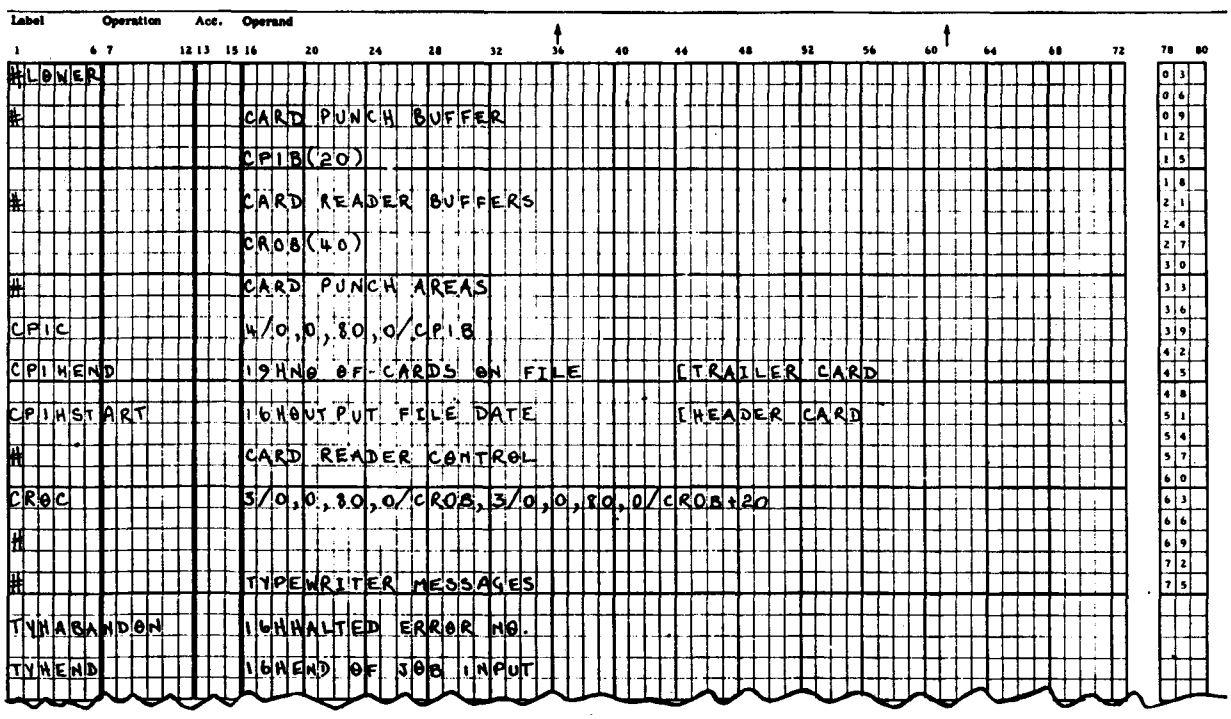
The next character is the unit number, though this may be omitted (except in the case of line printers) if no more than one peripheral of the same type is used. A numeric character is required in the third position for printer names to avoid clashing with the program labelling system.

The next character (third or fourth, that is) distinguishes the different areas within a peripheral group, as follows:

Peripheral buffers	B
Control and file definition areas	C
Headings, messages, etc.	H
Secondary, or Transfer, buffers from which information will be moved to the peripheral buffer, or vice versa.	T

Additional characters are at the programmer's discretion. Where possible these should be both systematic and meaningful. Names within groups, and the groups themselves, should be in alphabetical order.

Example



All other data names begin with either P for Preset or V for Variable. If the constants of a program are skilfully assigned, there should only be a few not in the peripheral areas. File area data not covered by the codes B,C,H, or T should take P or V in third or fourth character position. This would be applicable to page and line counts on the printer, for instance. Areas in Upper should use the character U before the P or V, to make them easily recognizable.

Labels

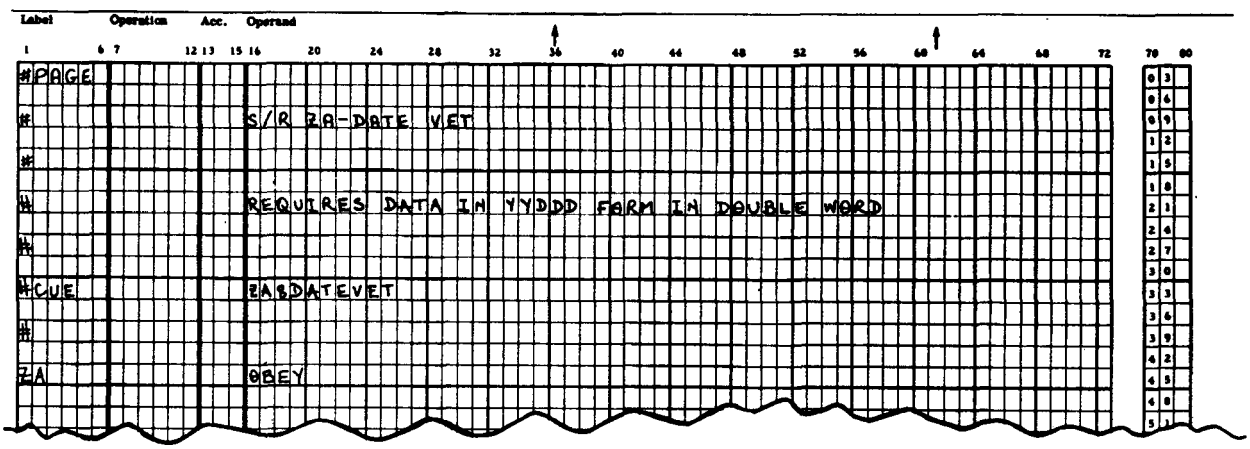
With the notable exception of FORTRAN, most ICL languages can use the labelling system of the flowcharting standards.

The program labelling method consists quite simply of placing an L in front of the flowchart box reference (which is wholly alphabetic). For example, for box ACE, the label will be LACE. Labels necessary between box starts will be the previous label with a numeric extension in the range 1 to 9 (or 0 to 9 if necessary), as LACE3. PLAN labels are restricted to five characters and this precludes the use of the two digit extension recommended in the flowcharting standards appendix.

Subroutines

Subroutines are named in the order in which they appear in the program, the first being subroutine ZA, the next ZB, in the sequence ZA to ZY, ZZA to ZZY, etc. The subroutine entry point should be CUEd (to put it in the consolidated list for easy reference) and annotated to indicate the function of the routine, the conditions required for entry, and any peculiarities.

Example



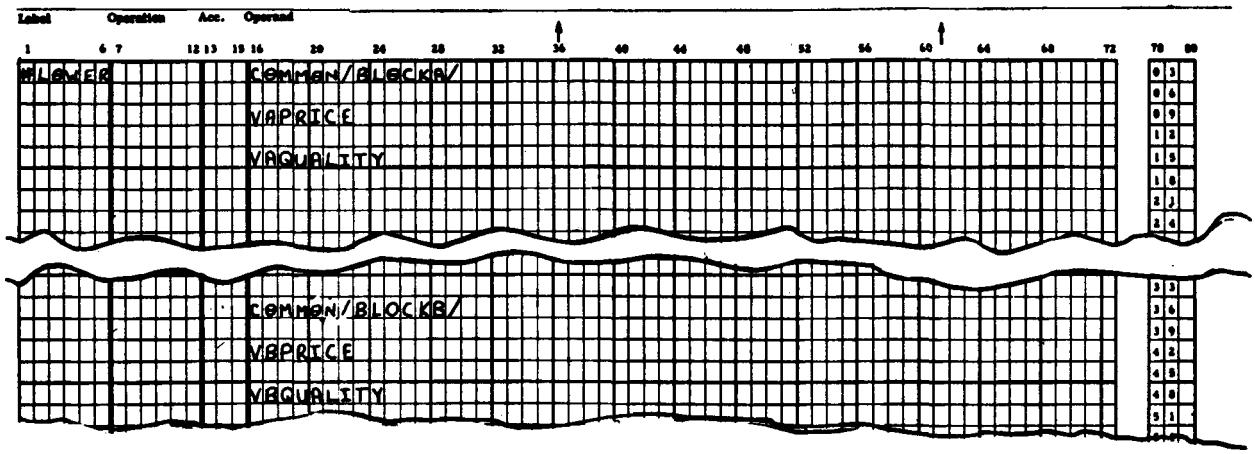
CUE names

CUEs should not normally refer to unlabelled coding. The name must have at its most significant end the label to which it refers, with possible extra information following a separator (the digit 8 is used in the above example).

COMMON names

These may be chosen freely but must be kept in strict alphabetical order. A systematic approach is recommended, for example, naming the COMMON blocks BLOCKA, BLOCKB, BLOCKC. The character which is systematically altered can appear in the program data names.

Example



This is particularly useful where there are a number of common areas, and it may not always be clear which area is referred to in the compiler listing at any given point.

ACCUMULATOR CONVENTIONS

In the interests of maintenance, a standard accumulator usage should be adopted, both in normal arithmetic and manipulation, and in subroutine linkage.

Normal usage

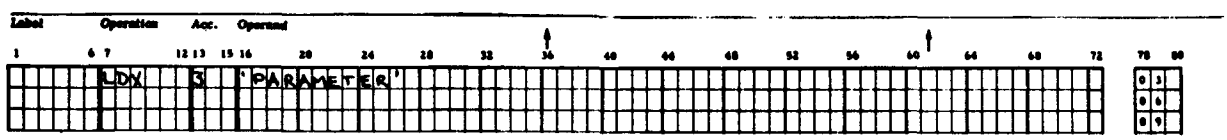
The following table outlines the generally accepted system of accumulator usage:

Accumulator	*Use*
0	CALL without parameters, otherwise free for arithmetic
1	CALL with parameters, reserve modifiers
2	Source modifier or sole modifier
3	Destination modifier, loading parameters of subroutines
4	Character testing, loading
5	Spare
6	Double-length values
7	Single-length values

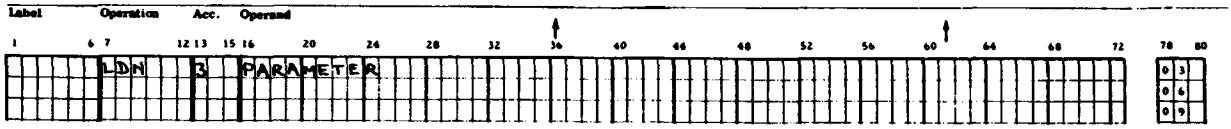
Note: The Input/Output Generator uses accumulator 2 for its CALLs.

Subroutine linkage

Wherever possible, the linking method adopted by the COBOL and FORTRAN compilers should be used. This will assist mixed language programming, as well as being a good method in its own right. The parameters are accessed by an OBEY instruction; the instructions to be OBEYed are listed after the CALL, and are of the form



or



where possible. This may not work in extended branch mode, however.

MONITOR

Bits 0 to 9 of word 30 should be reserved for monitoring purposes only.

Monitor identifiers should be in ascending sequence through the coding, and the sequence should be based on the physical order of the coding rather than the obeyed (that is the logical) order. The first monitor in a program using bit 1 of word 30 should be numbered 101 (or 1101 if accumulator suppression is required), the next monitor should be numbered 102 and so on. The first monitor using bit 2 should be numbered 201 (or 1201) the next 202, and so on.

Monitor can prove extremely useful during trials provided it is not used excessively. Selective monitoring, using a monitor switch subroutine, may be used here without limiting the printing of necessary information. The first requirement is the logical structuring of monitors; for example, bit 5 of the switch word may be used to monitor entries to and exits from segments, bit 6 to monitor subroutines, bit 7 to monitor logical breaks in the coding, bit 8 to print input/output areas, and so on. Then at strategic points in the program, for example, after reading a record, a call can be made to a monitor switch subroutine, which might switch selective monitors on every fourth record, or when a record of a particular type is being processed, and so on. The program should be designed for such monitoring. However, the details for the monitor switch subroutine can be left until the trials requirements become clear.

Monitor may also be used as an alternative to a post-mortem. A monitor subroutine which prints out only those areas of store the programmer is interested in can be given a specific entry point for this purpose.

MULTI-SEGMENT PROGRAMS

The average size of a segment (though this may depend on the complexity of the program) should be about 100 source instructions. This will allow reasonably short compile and recompile times, and keep the number of entries in the compiler tables small. Thus, long, meaningful data names can be used without fear of overflowing the table space.

It will be found useful in analyzing the program core print during trials to put all variables and presets in COMMON, as the block name appears in the consolidated list. It will also be found useful to give each segment a number, and use this as the penultimate digit of the monitor identifier within the corresponding segments. For example, segment 2 may have monitor identifiers 121, 122, 123, 221, 222, 1223.

EXAMPLES OF USE OF STANDARDS

The two examples that follow do not purport to be a good PLAN coding but were chosen as being relatively simple, and show many features of the standards in operation.

ICL 1900 Series PLAN coding sheet

Programmer J. BULL Sheet 1 of 6
Title PLAN JOB 2 Date 17 JULY 1969
Segment SGPRO Identity/Page 73 77

```
Label      Operation  Acc.  Operand
#STEER                      LIST,OBJECT,CHARGE=KJ018
#PROGRAM                    MMEP66/SGPRO
#PERMANENT                  SEGIN,SGOUT
#
#                           VARIABLES COMMON TO SEGIN AND SGPRO
#LOWER                      COMMON/COMNA/
                            VAPRTAB(350)          [PRICES IN PENCE
                            VATBATCH(2)           [INVOICE GROSS VALUE PER BATCH
#
#                           VARIABLES COMMON TO ALL SEGMENTS
#LOWER                      COMMON/COMNB/
                            VBRATECODE            [DISCOUNT CODE
                            VBREPTNO              [NO OF REPEAT LINES IN ORDER
                            VBSCATNO(111)         [SALES CATALOGUE NO
                            VBSQTY(110)           [QUANTITY
```

FORM 1/19167.000

ICL 1900 Series PLAN coding sheet

Programmer J. BULL Sheet 2 of 6
Title PLAN JOB 2 Date 17 JULY 1969
Segment SGPRO Identity/Page 73 77

```
Label      Operation  Acc.  Operand
#
#                           VARIABLES COMMON TO SGPRO AND SGOUT
#LOWER                      COMMON/COMNC/
                            VCDISC                [DISCOUNT VALUE
                            VCDISRT               [DISCOUNT RATE
                            VCGROSS(2)            [GROSS VALUE FOR EACH ORDER
                            VCNETT(2)             [NETT VALUE FOR EACH ORDER
                            VCPRICE(110)          [UNIT PRICE FOR EACH ORDER ITEM
                            VCVALUE(110)          [VALUE FOR EACH ORDER ITEM
#
#                           VARIABLES PARTICULAR TO THIS SEGMENT
#LOWER
                            VEDCNT(2)             [DISCOUNT VALUE WL
                            VEMULT                [CALCULATION WL
```

FORM 1/19167.000

ICL

1900 Series
PLAN
coding sheet

Programmer J. BULL | Sheet 3 of 6
Title PLAN JOB 2 | Date 17 JULY 1969
Segment SGPRO | Identity/Page 73 [ ] 77

Label	Operation	Acc.	Operand	
#				
#			CONSTANTS PARTICULAR TO THIS SEGMENT	
#LOWER				
PCONA			0	[DISCOUNT AS PERCENTAGE
			5	
			10	
			25	
PCONB			0	[DISCOUNT AS FRACTION
			0.05	
			0.10	
			0.25	
PCONC			0.50	[ROUNDING FRACTION

FORM 1/191067.000

ICL

1900 Series
PLAN
coding sheet

Programmer J. BULL | Sheet 4 of 6
Title PLAN JOB 2 | Date 17 JULY 1969
Segment SGPRO | Identity/Page 73 [ ] 77

Label	Operation	Acc.	Operand	
#PAGE				
#PROGRAM				
#			INITIALISE	
LP	STOZ		VCGROSS	[CLEAR GROSS
	STOZ		VCGROSS+1	
	LDCT	1	10	[MAX ITEMS PER ORDER = 10
#				
#			ACCUMULATE GROSS VALUE	
LQ	LDX	3	VBSCATNO(2)	[LOAD NEXT CATALOGUE NO
	BZE	3	LS	[IF ZERO THIS ORDER COMPLETE→
	SBN	3	1	[ADJUST FOR MODIFICATION
	LDX	6	VAPRTAB(3)	[FIND APPROPRIATE PRICE
	STO	6	VCPRICE(2)	[STORE PRICE
	MPY	6	VBSQTY(2)	[PRICE X QUANTITY
	STO	7	VCVALUE(2)	[STORE VALUE

FORM 1/191067.000

ICL 1900 Series PLAN coding sheet

Programmer J. BULL | Sheet 5 of 6
Title PLAN JOB 2 | Date 17 JULY 69
Segment SG | Identity/Page 73 [ ][ ][ ][ ] 77

Label	Operation	Acc.	Operand	
	ADS	67	VCGROSS	[ADD VALUE TO GROSS
LR	BUX	2	LQ	[ADJUST COUNT AND BRANCH IF MORE
#				
#			DISCOUNTS AND BATCH VALUE	
LS	LDX	67	VCGROSS	
	ADS	67	VATBATCH	[ADD GROSS TO BATCH TOTAL
	LDX	2	VBRATECODE	
	LDX	5	PCONA(2)	[LOAD PERCENTAGE DISCOUNT
	STO	5	VCDISRT	[STORE DISCOUNT RATE
	BZE	5	LSO	[BRANCH IF NO DISCOUNT
	LDX	5	PCONB(2)	[LOAD DISCOUNT AS FRACTION
	STO	5	VEMULT	[STORE IN MULTIPLIER
	LDX	66	VCGROSS	
	MPY	6	VEMULT	
	MPA	5	VEMULT	[GROSS X DISCOUNT FRACTION

FORM 1/161617.000

ICL 1900 Series PLAN coding sheet

Programmer J. BULL | Sheet 6 of 6
Title PLAN JOB 2 | Date 17 JULY 1969
Segment SGPRO | Identity/Page 73 [ ][ ][ ][ ] 77

Label	Operation	Acc.	Operand	
	BXL	7	PCONC,*+2	[BRANCH IF NO ROUNDING
	ADN	6	1	[ROUND RESULT
	STO	6	VCDISC	[STORE DISCOUNT VALUE
	STO	6	VEDCNT+1	
	BRN		LS1	
LSO	STOZ		VCDISC	[CLEAR FOR NO DISCOUNT
	STOZ		VEDCNT+1	
LS1	STOZ		VEDCNT	
	LDX	67	VCGROSS	[LOAD GROSS
	SBX	67	VEDCNT	[GROSS -DISCOUNT
	STO	67	VCNETT	[STORE IN NETT VALUE
	BRN		SGOUT	[PRINT SEGMENT
#END				
#FINISH				

FORM 1/161617.000

ICL 1900 Series PLAN coding sheet

Programmer J.BULL Sheet 1 of 6

Title P.T. TO M.T. COPY Date 16 JULY 1969

Segment Identity/Page 73 [ ][ ][ ][ ] 77

Label	Operation	Acc.	Operand	
#STEER			LIST,OBJECT,MONITOR,CHARGE-KTO18	
#PROGRAM			MMEP60	
#PERIPHERAL			TRO	
#				
#			BUFFERS	
#LOWER				
			MTBA(220)	[MT DATA BLOCK BUFFER
			MTBQ(20)	[MT QUALIFIER BLOCK BUFFER
			TRB(111)	[PAPER TAPE READER
#			CONTROL AREAS	
#LOWER				
MTCA			S/#1,0,220,0/MTBA	[DATA BLOCK CONTROL
MTCB			S/#400,0,12HINPUT▽DATA▽▽,0,6,7,0	[OPEN FILE CONTROL
MTCQ			S/#1,0,20,0/MTBQ	[QUALIFIER BLOCK CONTROL
TRC			O/#02,0,42,0/TRB	[PAPER TAPE READER

FORM 1/191067.000

ICL 1900 Series PLAN coding sheet

Programmer J.BULL Sheet 2 of 6

Title P.T. TO MT. COPY Date 16 JULY 1969

Segment Identity/Page 73 [ ][ ][ ][ ] 77

Label	Operation	Acc.	Operand	
#PAGE				
#PROGRAM				
#ENTRY			0	
#			OPEN MAGNETIC TAPE OUTPUT & OUTPUT HEADINGS	
#				
LA	PERI	0	MTCB	[OPEN MT
	WTM	0		[WRITE TAPE MARK
	LDN	2	0	[ZEROIZE RECORD LENGTH COUNTER
	LDN	5	0	[ZEROIZE BLOCK COUNTER
	LDX	4	'#400000002'	[SET UP QUALIFIER BLOCK
	STO	4	MTBQ	
	LDN	4	220	[BLOCK LENGTH MAXIMUM
	STO	4	MTBQ+1	

FORM 1/191067.000

ICL

1900 Series PLAN coding sheet

Programmer J. BULL | Sheet 3 of 6

Title P.T. TO M.T. COPY | Date 16 JULY 1969

Segment | Identity/Page 73 [ ][ ][ ][ ] 77

Label	Operation	Acc.	Operand	
	STO		MTBQ+2	CCLEAR REMINDER OF BLOCK
	LDN	6	MTBQ+2	
	LDN	7	MTBQ+3	
	MOVE	4	17	
	PERI	0	MTCQ	COUTPUT QUALIFIER BLOCK
#				
LB	ON	4	1	CSET SWITCH FOR MONITOR USE
#				
#				READ AND PROCESS PAPER TAPE RECORD
LC	PERI	0	TRC	CREAD PAPER TAPE RECORD
	SUSBY		TRO	CSUSPEND PROGRAM
#MONITOR			1/101	CPRINT PT RECORD
			111/TRB	

FORM 1/101617.000

ICL

1900 Series PLAN coding sheet

Programmer J. BULL | Sheet 4 of 6

Title P.T. To M.T. COPY | Date 16 JULY 1969

Segment | Identity/Page 73 [ ][ ][ ][ ] 77

Label	Operation	Acc.	Operand	
LD	LDX	4	TRB+1	CSECOND WORD OF PT RECORD
	SRL	4	12	CCHS. 5&6 OF PT RECORD
	BXE	4	'#7632',LH	CTEST DELTA,NEWLINE→END OF FILE
LE	LDN	6	TRB	CADDRESS OF RECORD START INPUT
	LDN	7	MTBA+1(2)	CADDRESS OF RECORD START OUTPUT
	MOVE	6	10	CMOVE RECORD
	LDN	4	11	CRECORD LENGTH
	STO	4	MTBA(2)	CSTORE IN OUTPUT AREA
LF	ADX	2	4	CSTEP RECORD LENGTH COUNTER
	BXU	2	'220',LC	CTEST BLOCK FULL, IF NOT→
#				
#				OUTPUT BLOCK, STEP BLOCK COUNT
LG	PERI	0	MTCA	COUTPUT BLOCK
	LDN	2	0	CZEROIZE RECORD LENGTH COUNTER
	ADN	5	1	CSTEP BLOCK COUNTER

FORM 1/101617.000

ICL

1900 Series
PLAN
coding sheet

Programmer J. BULL Sheet 5 of 6

Title P.T. TO M.T. COPY Date 16 JULY 1969

Segment Identity/Page 73 [ ] 77

Label	Operation	Acc.	Operand	
	SUSAY		MTO	[SUSPEND PROGRAM
	BRN		LC	[READ NEXT RECORD→
#				
#			END OF INPUT FILE	
LH	BZE	2	LI	[NO PART-BLOCK LEFT TO OUTPUT→
	LDN	4	2	[END RECORD. LENGTH
	STO	4	MTBA(2)	[STORE IN OUTPUT AREA
	LDX	4	'LH*****'	[END RECORD. IDENTIFIER
	STO	4	MTBA+1(2)	
	PERI	0	MTCA	[OUTPUT BLOCK
	ADN	5	1	[STEP CLOCK COUNTER

FORM 1/191617.000

ICL

1900 Series
PLAN
coding sheet

Programmer J. BULL Sheet 6 of 6

Title P.T. TO M.T. COPY Date 16 JULY 1969

Segment Identity/Page 73 [ ] 77

Label	Operation	Acc.	Operand	
#				
#			WRITE END BLOCKS AND CLOSE PROGRAM	
LI	WTM	0		[WRITE TAPE MARK
	LDX	4	'#40000000'	[SET UP QUALIFIER BLOCK
	STO	4	MTBQ	
	STO	5	MTBQ+1	[NO OF DATA BLOCKS
	PERI	0	MTCQ	[OUTPUT QUALIFIER BLOCK
	UNL	0		[UNLOAD MAGNETIC TAPE
	DEL		2HOK	[DELETE AND TYPE OK
#END				
#FINISH				

FORM 1/191617.000

Appendix 14 Usercode standards and techniques

The standards that follow are intended to assist the programmer in laying out his program so that it may be understood by another programmer more easily than if no convention whatever had been followed. Clarity of both layout and intention is the aim. It is hoped that serious thought will be given to the recommendations herein, and that the techniques described will prove helpful. References to the 4–40 may be taken as referring to 4–50 and 4–70 as well.

STANDARDS

Writing

Characters should be clearly legible. At least every other line on the coding sheet should be left blank, to enable amendments to be easily inserted; even if amendments are not anticipated, the blanks greatly increase the legibility of the coding.

Layout

Free use should be made of the EJECT, SPACE, and TITLE directives, not only to separate routines or groups of constants, but also to separate the many logical units which make up even a small subroutine. SPACE directives should average about one for every ten instructions. All routines and subroutines should have their own TITLEs. Any new logical section in a routine should start with an EJECT, and these should average two or three per routine.

Comments

The importance of comments in low-level language programs cannot be over-stressed. Not only do they greatly assist others to understand the program (as well as assisting the author who may forget his intentions), but also relate the coding to the flowchart, and minimize unclear thinking. Comments should average about one for every two lines of coding; one for one is even better. Comments must be meaningful. Declarations of constants, work areas and so on should bear comments describing the intended use of the areas. Switches in particular need a thorough explanation.

Numbering

It is recommended that source packs be punched with sequence numbers. Apart from the protection offered against card-droppers, there is one further advantage. If the first two digits of the eight digit sequence number are given the module number, so that all cards in module 5 have sequence numbers beginning 05..., then later amendments accidentally made to the wrong module on a trials disc will not adversely affect it. Renumbering, which would defeat the purpose of the exercise, can be avoided by initially numbering the cards in units of one hundred.

Symbolic data names

The standard method of naming is designed to enable the programmer to trace data references, and in some cases to suggest the contents of the location. In no way does this method replace the use of comments. The first three characters of all data names describe the type and position of the data; the remainder should be used to provide a meaningful description.

Character	*Function*
1st character	Type code: This will be one of the following: W for work areas N for instruction constants S for S-type constants A for address constants

Character	*Function*
	V for V-type constants Y for Y-type constants C for other constants (Areas which are given initial values, but are intended as work areas have type code W as the first character.) For files the type code is considered as a file letter, and every file is allocated a letter in the range B, D to H, J, K, P, Q, R, T, U, X. All file declarations should have a comment indicating the file letter allocated. Work areas and dummy sections associated with files should use the file letter rather than the type code.
2nd character	Module code: This is an alphanumeric code in the range A to Y, 1 to 9. The letter A should be designated to the control module.
3rd character	Group code: This is an alphanumeric code in the range A to Z, 1 to 9. A group of constants can be subdivided (by using the SPACE directive) into groups of ten to twenty constants, each of which is given a different code. Constants should be in alphabetical order so all constants of the same type will be grouped together. (SYS ..... for S-type constants should be avoided.)
MEANINGFUL DESCRIPTION 4th to 8th characters	
For address constants V-type constants S-type constants Y-type constants	Either (a) a three-character reference code of the location to which the constant refers, followed by one or two digits to make the name unique. or (b) the last five characters of the name to which the constant refers.
For instruction constants	The Usercode mnemonic of the instruction, followed by one or two digits to make the name unique.
For other constants	The type of data code used in the define constant instruction followed by the constant value, or another description if the value is too long, for example, CABP23 for a constant defined as P'23'. If the type and constant value are used for these five characters, and the constant is changed, then the name of the constant (both the declaration and the usage) must be changed.
For work areas	A five character description of the purpose of the area should be used. Similar names in the same program should be avoided.
For input/output areas	A single digit should be used to indicate the area within a file. Characters 5 to 8 are not specified. *Example* EAA1 for IOAREA1 for third file and EAA2 for IOAREA2 for third file.

Labels

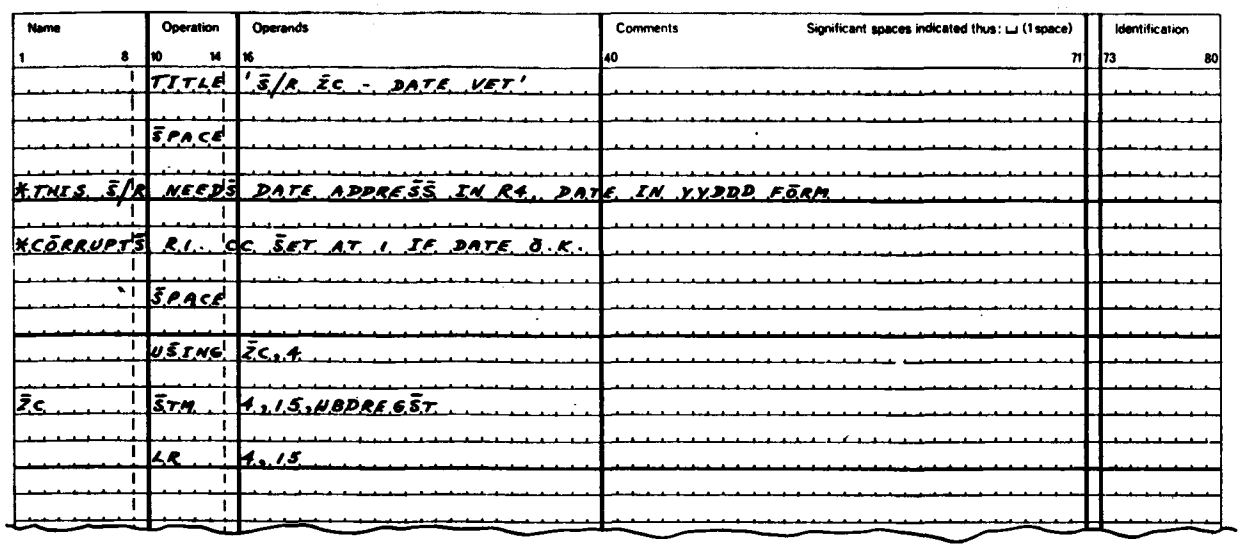
The first line of coding for a box on a flowchart must have a label in the form L followed by a flowchart box reference. For example, box ACE will be labelled LACE. Labels necessary within a box will be the previous label with a numeric extension in the range 01 to 99, as LACE03.

Subroutines

Subroutines are named from the beginning of the program, the first being subroutine ZA, the next ZB, in the sequence ZA to ZY, ZZA to ZZY, and so on.

The subroutine entry point should be annotated to indicate the function of the routine, the conditions required for entry, and any peculiarities.

Example



Name (1–8)	Operation (10–14)	Operands (16)	Comments (40–71) Significant spaces indicated thus: ␣ (1 space)	Identification (73–80)
	TITLE	'S/R ZC - DATE VET'		
	SPACE			
*THIS S/R	NEEDS	DATE ADDRESS IN R4, DATE IN YYDDD FORM		
*CORRUPTS	R1.	CC SET AT 1 IF DATE O.K.		
	SPACE			
	USING	ZC,4		
ZC	STM	4,15,12BDRE6ST		
	LR	4,15		

Modules, CSECTs

These labels are derived as follows:

1st character	Type code: M
2nd character	Module code: This is an alphanumeric code in the range A to Y, 1 to 9. The letter A should be designated to the control module.
3rd to 7th characters	Meaningful description: This is the five character program name.
8th character	The final character is A for the START directive: for subsequent CSECTs, and ENTRY names which are not subroutine or data names, the final character is in the range B to Z, 1 to 9.

Note: The 5E regime requires a module name to be unique in its first six characters. It is therefore suggested that a zero is added to E-level program names for use in the module name. If this method is adopted, then J-level programs need not be restricted.

Macro names

The macro name standards must be strictly followed to avoid confusion arising from possible duplication.

Macros written for or by the user should be in the form

LLLdd

where

LLL is the user's installation code.

dd is a serial number of which at least one character is numeric.

Register usage

The following system is based on the method of calling subroutines used by the majority of System 4 compilers. It has several advantages over the system which was used in the past in that the same registers are used for calling both Usercode subroutines and subroutines in any other language; the link registers used are those which are corrupted by the File Control Package (FCP), so that non-corrupted registers are free for holding long-term totals, etc., and complete compatibility is assured with the subroutine calling and linking macros XCALL, XNTER, XEXIT.

Basically, subroutines are called using register 15, the return address being in register 14.

Example

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 Significant spaces indicated thus: ␣ (1 space) 71	73 80
	L	15,=A(SUB)		
	BALR	14,15		

Any parameters are passed using register 1.

Example

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 Significant spaces indicated thus: ␣ (1 space) 71	73 80
	LA	14,LADI		
	L	15,=A(SUB)		
	BALR	1,15		
WADPAR1	DC			
WADPAR2	DC			
WADPAR3	DC			
LADI	MVC			

Register 13 may be used to hold the address of a work area which the called subroutine may use, and register 12 to replace register 1 for parameter addressing in those routines (mostly FCP) which corrupt register 1. The subroutine base register is register 4, unless the subroutine never calls another, when register 15 may be sufficient.

Thus, among the first instructions in a subroutine which calls another, or otherwise corrupts register 15, will normally be

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 Significant spaces indicated thus: ␣ (1 space) 71	73 80
	LR	4,15	S/R ZB BASE REGISTER	
	USING	ZB,4		
	LR	12,1	FOR PARAMETER ADDRESSING	

The last instruction is inserted for those subroutines which will corrupt register 1.

The recommended use of registers is shown in the table.

4–30	*General register*	*4–40*
Timer register, must not be used	0	Corrupted by FCP. Use carefully for counting, etc., or with R1 as even-odd pair.
Corrupted by COMRG. Use as parameter base register, and for general short-term arithmetic.	1	Corrupted by TRT, EDMK, FCP. Use as parameter base register, and for general short-term arithmetic, etc.
General purpose.	2	Corrupted by TRT. General purpose.
Base register for data module or control routine.	3	Base register for data module or control routine.
Base register for subroutines.	4	Base register for subroutines.
Chiefly IOREG, VARBLD, etc. Any left over may be used for indexing, arithmetic, etc.	5 to 9	Chiefly IOREG, VARBLD, etc. Any left over may be used for indexing, arithmetic, etc.
Use for STXIT routines, otherwise as for 5 to 9.	10,11	Use for STIXT routines, otherwise as for 5 to 9.
Standby register for parameters in subroutines where R1 is corrupted. Otherwise general purpose.	12	Standby register for parameters in subroutines where R1 is corrupted. Otherwise general purpose.
Register to address general purpose work area for subroutines. Otherwise general purpose.	13	Register to address general purpose work area for subroutines. Otherwise general purpose.
Corrupted by FCP. Link register for subroutines.	14	Corrupted by FCP. Link register for subroutines.
Corrupted by FCP. Branch register for subroutines	15	Corrupted by FCP. Branch register for subroutines.

The following table summarizes these rules by usage; the order of preference of registers is from left to right. Registers specified are assumed to be free. The use of register 0 is not advocated for 4–30.

	General register	
Usage	*Short term usage*	*Long term usage*
IOREG, VARBLD, etc.	5,6,7,8,9,10,11, 12,13,2	5,6,7,8,9,10,11
Indexing	11,10,9,8,7,6,5, 2,12,13	11,10,9,8,7,6,5
Arithmetic in pairs	0,1;14,15;12,13; 10,11;8,9;6,7.	10,11;8,9;6,7.
Arithmetic singly	0,1,14,15,2,13, 12,11,10,9,8,7 6,5	11,10,9,8,7,6,5
Base register (control routine)	3	3
Base register (subroutine)	15,4,	4
Parameter register	1,12,	12
Work area addressing	13	13
Branch register	15	15
Link register	14	14

GENERAL TECHNIQUES

Module size

One of the main advantages of programming in modules is that the amount of re-compilation during trials is reduced. This is best achieved with small modules. Small modules also provide the advantages of requiring few people to be involved with the coding and compilation of each module, and of avoiding over-running a base register. The recommended module size is about 4K bytes. Cases could arise, however, where modules may advantageously be smaller than this; thus the figure of 4K should be treated more as a current acceptable norm: the principles are more important than the figure itself.

Module linkage, program structure

The system described here is designed for medium to large size programs; smaller programs may not require all the features described. The main aim is to keep the modules of the program fairly small, with standardized linkages, thereby aiding comprehension and increasing the speed of trials.

CONTROL/DATA MODULE

The control module will normally use base register 3, which must never be altered. After BALR 3,0 are all useful constants, storage and work areas, including a special hierarchical register store. For very large programs, this will probably not leave room for many instructions under register 3; in this case, the module becomes a data module, and instructions are written in a separate control module, with base register 4.

Example

For medium programs, the control module will be:

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 71	73 80
	USING	LA+2,3		
	CNOP	6,8		
LA	BALR	3,0		
	B	LA1		
	SPACE			
AAASQUA	DC	A(NADSQUA)	ADDRESS OF	
	EJECT			
LA1	NVC			

For large programs, the control module will be:

Name	Operation	Operands	Comments (Significant spaces indicated thus: ␣ (1 space))	Identification
1 8	10 14	16	40 71	73 80
	USING	LA,2,4		
LA	BALR	4,0		
	L	3,=V(AAASQUA)		
	USING	CAADATA,3		
CAADATA	DSECT		REFERRING TO CONSTANTS IN	
*			DATA MOD	
AAASQUA	DS	F		
MARIIIA	CSECT			
LA1	MVC			

The data module will be:

	USING	AAASQUA,3	
	CNOP	0,8	
AAASQUA	DC	A(MADSQUA)	

Note: If several large areas are needed in the data module, addressing problems can be overcome to some extent by interleaving them. Though this is not possible where a work area must be a continuous whole, it is perfectly sound for tables, arrays, etc.

Example

Name	Operation	Operands	Comments (Significant spaces indicated thus: ␣ (1 space))	Identification
1 8	10 14	16	40 71	73 80
A	DS	2000F		
B	DS	2000F		
C	DS	2000F		

These work areas may be expressed as:

Name	Operation	Operands	Comments (Significant spaces indicated thus: ␣ (1 space))	Identification
1 8	10 14	16	40 71	73 80
A	DS	6000F		
B	EQU	A+4		
C	EQU	A+8		

This would give a store area A_1 B_1 C_1 A_2 B_2 C_2 – – –

The processing routines of the program are called using register 4 as the base register. Before this register is loaded, all registers are stored in the hierarchical register store. It may not be necessary to store all the registers, but it is probably better to do this, since more registers may be required than is at first anticipated.

THE HIERARCHICAL REGISTER STORE

The hierarchical register store consists of a series of 64 byte areas, full-word aligned, one area for each level of subroutine nesting. Routines at the first level store their registers in the first area, and so on.

Example

For module 1

Name (1–8)	Operation (10–14)	Operands (16)	Comments (40–71)	Identification (73–80)
	BALR	3,0		
VADZA	DC	V(ZA)	MAIN PROCESSING ROUTINE	
VADZB	DC	V(ZB)	DATE VET ROUTINE	
WAERSTO1	DS	16F		
WAERSTO2	DS	16F		
	L	15,VADZA		
	BALR	14,15		

For module 2

Name (1–8)	Operation (10–14)	Operands (16)	Comments (40–71)	Identification (73–80)
CBADATA	DSECT		TO REFER TO CONSTANTS IN MOD 1	
VADZA	DS	F		
VADZB	DS	F	DATE VET ROUTINE	
WAERSTO1	DS	16F	H REGISTER STORE	
WAERSTO2	DS	16F		
MBR1111A	CSECT			
	USING	ZA,4		
ZA	STM	0,15,WAERSTO1		
	LR	4,15		
	L	15,VADZB		
	BALR	14,15		
	LM	0,15,WAERSTO1		
	BR	14		

The subroutine ZB starts

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	USING	ZB,4			
ZB	STM	0,15,WAREST02			
	LR	1,15			

On 4–40, since register 3 must never be altered, it is possible to use STM 4, 2 instead of STM 0,15, saving one word on each storage location. Since, on 4–30, register 0 is the timing register, it is possible to use STM 1, 15. 4–30 must use LA and B instead of LR and BR.

DYNAMIC WORKING STORE ALLOCATION

Particularly for programs where store is at a premium, a rather neater and more flexible method for storing registers and assigning working store may be used, at the expense of the full-time use of register 13. In this method, none of the routines or subroutines have their own working store, all this being at the end of the data module. The start address of free working store is held in register 13, and all routines have a DSECT using base register 13 instead of real working store. Thus, if the main routine uses 400 bytes of working store, then before it calls its first subroutine, the instruction

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	LA	13,400(0,13)			

is performed, to set up the free working store for the subroutine; when control is returned, 400 must be subtracted from register 13 to address the work area correctly again. Similarly, if the subroutine itself uses 104 bytes of working store, then before its first subroutine is called, the instruction

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	LA	13,104(0,13)			

must be performed, and 104 subtracted on return. It will be seen that the amount of working store needed is not the sum of the working stores needed by each subroutine, but the maximum of the store requirements for each possible path through the program.

Care must be taken that register 13 is incremented only in units of 8, so that the registers can be stored at location 0(13), and any DSECT is correctly aligned.

Note: If the data module is at the highest store location, the amount of working storage needed can be allocated in the //␣STORE card or during composition. This can be useful if the amount of working store varies significantly from one run-type to another.

THE DSECT

The chief disadvantage of using small, 4K byte modules is that each module must possess a DSECT with which to refer to the constants and work areas in the data or control module. To overcome any difficulty, all constants (including the first branch instruction, if there is one) should be submitted in a macro to the macro library. Then it is only necessary to write the macro name to generate the whole DSECT; amendments are made direct to the macro library, and incorporated into the several modules of the program by simply re-compiling. The macro should be carefully defined to give the length of constants where necessary.

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	DC	C'HEADING'			

will cause the translator to move the location counter only one byte in its calculation of DSECT displacements. The correct instruction should be:

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	DC	CL7'HEADING'			

which in a DSECT is translated as

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	DS	CL7			

Example

Submitted to the macro library might be:

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	MACRO				
&NAME	XYZIA				
	B	LA0,1			
	SPACE				
AAASEQA	DC	A(ADSEQA)	ADDRESS OF		
	EJECT				
	MEND				

The control module can now be written:

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 71	73 80
	USING	LA,2,3		
	CNOP	6,8		
LA	BALR	3,0		
	XYZIA			
LAO1	MVC			

The next module of the program might be written:

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 71	73 80
	USING	CBADATA,3		
CBADATA	DSECT		CONSTANTS	
	XYZIA			
MBR1111A	CSECT			

This example shows a clear saving of effort and errors.

The amount of store may be minimized by organizing the constants, work areas, etc., in the order:

Double words

Full words

Half words

Bytes

though it is doubtful if the amount of store saved is worth the inconvenience of not having the constants in alphabetical order.

TRIALS

To save frequent re-compilation during trials, it might be found useful to have a patch area under base register 3. Any large sections of coding that have to be inserted can be put here by REP cards, and branched to from any part of the program. REP cards should not, however, become a permanent feature of a program, but be used only to avoid recompilation temporarily.

Address calculations

All arithmetic errors that occur during the writing of a program are dependent on the programmer's using calculations on numbers. It follows that if the programmer avoids these, and leaves as much of the arithmetic as possible to the Translator, he will produce a correct program more quickly. Also, he will find the program easier to amend, and easier to understand. Whilst it is not possible to give general rules on the avoidance of calculations, a few examples of the technique follow, to give some idea of the possibilities.

SELF-DEFINING VALUES

Not only are decimal numbers self-defining, but also expressions such as X'26' and C'?'. Thus, instead of a table drawn up as:

```
WACTABLE DC    64C'*'
         DC    C' '
         DC    9C'*'
         DC    C'£.<(+|&'
         DC    9C'*'
```

to translate unprintable characters to *, it is much easier and safer to avoid calculations with the following coding:

```
WACTABLE DC    256C'*'
         ORG   WACTABLE+C' '
         DC    C' '
         ORG   WACTABLE+C'£'
         DC    C'£.<(+|&'
         ORG   WACTABLE+C'!'
```

Also, the translation that is required is very much clearer.

Similarly, if a table has been drawn up to translate from the character A onwards, then:

```
         TR    WABLINE(8),WACTABLE - C'A'
```

is clearly superior to:

```
         TR    WABLINE(8),WACTABLE - 193
```

Again, the load address instruction can be used to load a one byte character into a register.

```
         LA    1,C'+'
```

ABSOLUTE EXPRESSIONS

Absolute expressions are permitted for lengths, displacements, etc., and may be used freely to allow the translator to perform the arithmetic.

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 Significant spaces indicated thus: ␣ (1 space) 71	73 80
	MVC	A(4*2+3),B		

is perfectly legal. A maximum of three operators is permitted.

LENGTH ATTRIBUTE

The length attribute of a symbol is written in the form L' *n*.

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 Significant spaces indicated thus: ␣ (1 space) 71	73 80
A	DS	CL7		

L'A has the value 7.

If all constants have been defined with lengths, then the length attribute may be used to advantage, as in the following coding.

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 Significant spaces indicated thus: ␣ (1 space) 71	73 80
B	DS	CL4		
	MVC	A(4),B		

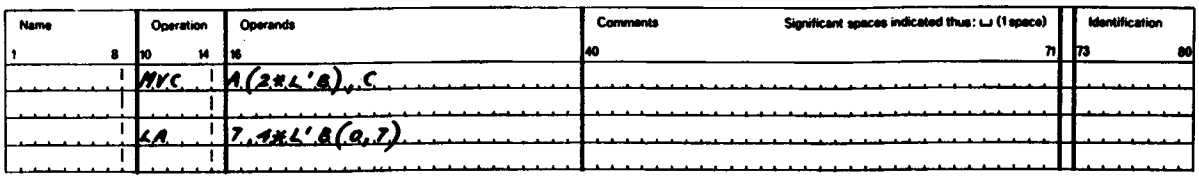
In this way, any change to the length of B will not effect the subsequent coding.

The length attribute may also be used to advantage where it is required that one work area is two or three times the length of another

Example

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 Significant spaces indicated thus: ␣ (1 space) 71	73 80
A	DS	1000H		
B	EQU	*		
	ORG	*+2(B-A)		
C	EQU	*		
	ORG	*+3(B-A)		

Expressions may also be written in the following form. The second instruction may be found useful in loops, for instance.



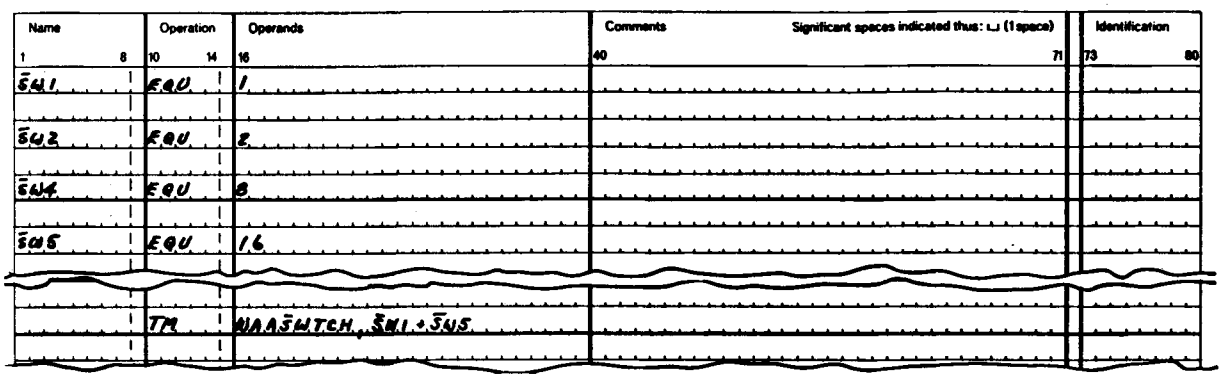
Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	MVC	A(2*L'B),C			
	LA	7,1*L'B(0,7)			

EQU DIRECTIVE

The EQU directive can be used to give meaningful names to mask fields which are otherwise not meaningful.

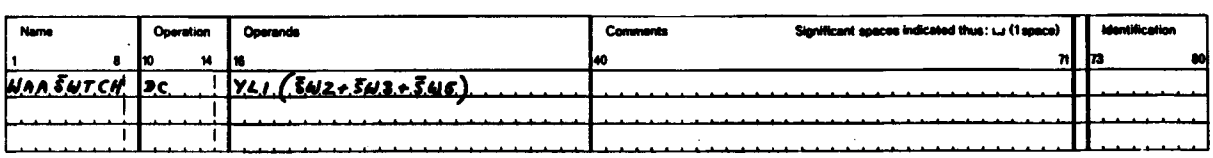
Example

To test if switches 1 and 5 are set, the following coding may be used.



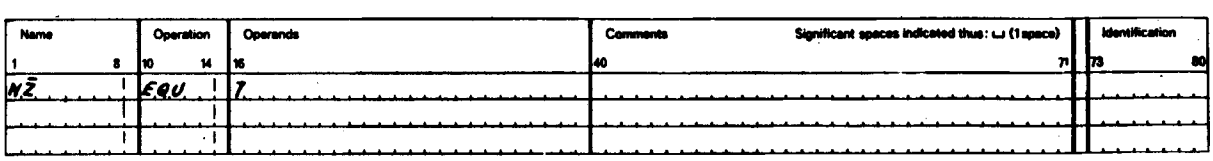
Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
SW1	EQU	1			
SW2	EQU	2			
SW4	EQU	8			
SW5	EQU	16			
	TM	WAASWITCH,SW1+SW5			

Switches may be set up by the following coding.



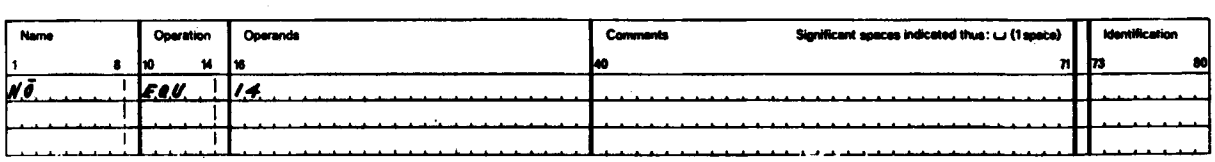
Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
WAASWITCH	DC	YL1(SW2+SW3+SW6)			

Again there are a few condition code masks for which there is no extended mnemonic, and which may be symbolically defined.



Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
NZ	EQU	7			

for branch on non-zero (including overflow) or branch not all zeros (after TM).



Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
NO	EQU	14			

for branch on no overflow, or not all ones (after TM).

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
NM	EQU	11			

for branch on not minus (including overflow) or not mixed (after TM).

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
NP	EQU	13			

for branch on not plus (includes overflow).

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	TM	HAASWITCH, SW4			
	BC	NO, LABEL	IF SW4 NOT SET		

The EQU directive may further be used as a guard against mis-punching.

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
I	EQU	1			
O	EQU	0			

These directives would cause mis-punches to be correctly translated.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	MVC	A(I), B			
	LA	6, 6(O, 6)			

The following directives should be used with care.

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
S	EQU	5			
Z	EQU	2			

S– and Z– type constants would be mis-translated.

CLEARING

The most efficient way of clearing a register is to subtract it from itself.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	SLR	4,4			

To clear an area of core (maximum size 256 bytes), an exclusive OR can be performed on that area.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	XC	STORE1,STORE1			

To clear a single byte of core, use the Move Immediate instruction to store a byte of zeros.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	MVI	BYTE1,0			

TESTING

In order to test the state of a register the condition code can be set by loading the register into itself with the load and test register instruction. It should be noted, however, that most load instructions do not affect the condition code.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	LTR	4,4			

To test an area of core for zero or non zero, the condition code can be set by performing a logical OR on that area.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	OC	STORE1,STORE1			

To test a single byte in core for zeros, ones or a mixture, the test under mask instruction with a mask of all ones (that is, X'FF') should be used.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	TM	BYTE1,X'FF'			

However, CLI is faster than TM and should be used where possible.

EXCHANGING

If the contents of two registers or two areas of core are to be exchanged without the use of any further register or area of core, then three exclusive OR instructions should be used.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	XR	4,5			
	XR	5,4			
	XR	4,5			

or

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	XC	STORE1,STORE2			
	XC	STORE2,STORE1			
	XC	STORE1,STORE2			

If the two halves of a single byte are to be exchanged, it is only necessary to use the PACK (or UNPK) instruction.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	PACK	BYTE1,BYTE1			

ALTERING BRANCH INSTRUCTIONS AND SWITCHING

If an unconditional branch is to be altered to a NOP instruction, then a logical AND with an immediate constant to the byte following the address of the unconditional branch should be used. For example, if the location of the unconditional branch is BRANCH, then the instruction should be

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	MI	BRANCH+1,X'0F'			

If the MVI instruction were used the X_2 field of the branch would also be zeroized.

If an NOP instruction is to be altered to a branch instruction use an inclusive OR instruction with an immediate constant.

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	OI	NOBRANCH+1,X'F0'			

It should be noticed that these instructions can also be considered as program switches, since a test mask instruction can be used to test the condition code mask field of the branch instruction. It is not necessary to define a separate switch simply because more than one branch point depends on the switch setting.

Subroutines may be entered at different points by use of the BAL instruction, which allows multiple switching.

Example

Calling module

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	L	15,=V(EA)			
	BAL	14,16(0,15)			

Subroutine

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	USING	EA,15			
EA	B	LFA			
	B	LFB			
	B	LFC			
	B	LFD			

Multi-way switches are best handled using S-type constants.

Example

A switch is to be made on the run-type which is held in a half-word binary location WAARNTP. Register 5, the base register for S-type constants, is used here as an index register.

Name	Operation	Operands	Comments (Significant spaces indicated thus: ␣ (1 space))	Identification
	L	5,=A(SAASWTAB)	A(S-CONSTANTS) FOR SWITCH	
	AH	5,WAARNTP	TWICE BECAUSE S-TYPE CONSTANTS	
	AH	5,WAARNTP	2 BYTES LONG	
	MVC	LB1+2(2),0(5)	S-CONSTANT TO BRANCH INSTRUCTION	
LB1	B	LA	LA IS DUMMY ADDRESS	
	USING	SAASWTAB,5		
SAASWTAB	DC	S(LC)	FOR RUN-TYPE 0	
	DC	S(LE)	FOR RUN-TYPE 1	
	DC	S(LG)	FOR RUN-TYPE 2	
	DC	S(LI)	FOR RUN-TYPE 3	
	DC	S(LK)	FOR RUN-TYPE 4	
	DROP	5		

ROTATION AND SHIFTING

The contents of a register may be rotated by the use of a double shift instruction followed by inclusive OR instruction.

Example

The contents of register 5 are to be rotated by 15 places.

Name	Operation	Operands	Comments (Significant spaces indicated thus: ␣ (1 space))	Identification
	SR	4,4	CLEARS REGISTER 4	
	SLDL	4,15	SHIFTS REGISTERS 4 AND 5 BY	
*			15 PLACES	
	OR	5,4	INCLUSIVE OR BACK	

If the contents of a small table of consecutive words in core are to be rotated, then multiple load and store instructions (providing there are sufficient registers available) should be used.

Example

Name	Operation	Operands	Comments (Significant spaces indicated thus: ␣ (1 space))	Identification
	LM	1,7,STORE1		
	LR	8,1		
	STM	2,8,STORE1		

This example used an extra register (register 8), but saves two instruction bytes by using the LR instruction in place of an L or ST instruction.

It should be emphasized that the MVC instruction can only be used to shift an area of core to the left, because of the sequence in which bytes are moved. However, to shift decimal numbers to the right, the MVO instruction may be used, but only for odd numbers of places; for even numbers moves are made by two odd number moves in succession.

Example

The contents of WAANUMB are to be altered from 01234F to 00123F.

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	MVO	WAANUMB(3),WAANUMB(2)			

INDEXING

If an index is to be loaded with a positive constant in the range 1–4095, use the load address instruction.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	LA	4,ADDRESS			

To increment an index by a positive constant in the range 1–4095, the load address instruction should be used.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	LA	4,INCREM(4)			

To place the sum of two index registers into a third register, the load address instruction is used.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	LA	6,0(4,5)			

Note: The displacement may also be used to increase the contents of the third index register by a small positive amount at the same time.

Example

To increase the contents of the third index register by 12.

Name	Operation	Operands	Comments	Significant spaces indicated thus: ⌴ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	LA	6,12(4,6)			

Care should be taken when using the load half-word instruction to load an index. The most significant bit is taken as the sign and generated through the left half of the register. It is therefore advisable to use an AND instruction after the load half-word instruction to avoid any problems that might arise.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ⌴ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	LH	4,HWORD1			
	N	4,=X'0000FFFF'			

There is no real difference between a base register and an index register. If care is taken, base register modification can give the effect of indexing for SS instructions, or can provide double indexing for RX instructions which may be used for processing 2-dimensional arrays etc.

BINARY ARITHMETIC

When concerned with only the least significant 24 bits (or less), any of the indexing techniques may be used.

When concerned with only the least significant 12 bits (or less), the load address instruction may be used to subtract positive constants in the range 1 to 4095.

Example

The amount 548 is to be subtracted from the least significant 12 bits of register 4.

Name	Operation	Operands	Comments	Significant spaces indicated thus: ⌴ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	LA	4,4096-548(4)			

It should be noted that a carry into the 13th bit (bit 19) will occur, so that no more than 12 bits of the results can be used.

To subtract 1 from a register, use the branch on count in register instruction.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ⌴ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	BCTR	4,0			

This can be useful to adjust a length value before executing, with EX, a logical SS instruction.

In order to conserve space, half-word constants and variables should be used in place of full-word constants wherever possible.

The division and multiplication by any power of two should be accomplished by shift instructions.

CHARACTER MANIPULATION

Care should be taken when using the load half-word instruction on a 2-byte character field in a register as described in *Indexing*, above.

Care should also be taken when using the insert character instruction (IC) without first clearing the register, as the instruction will leave the contents of bits 0–28 unaltered.

Load address can also be used to load a one-byte character constant into a register.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	LA	4,C'%'			

The immediate instructions MVI, CLI etc. are very useful in single character manipulation.

An area of core may be filled with a particular character by using the following sequence of instructions:

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	MVI	STORE1,C'Z'			
	MVC	STORE1+1(L'STORE1-1),STORE1			

or

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	MVC	STORE1(1),CHAR			
	MVC	STORE1+1(L'STORE1-1),STORE1			

MISCELLANEOUS

Multiple-precision fixed point arithmetic may be accomplished by the use of the add logical and subtract logical instructions on all but the most significant bit. These provide exactly the same results as the add and subtract instructions, but avoid the possibility of an overflow interrupt.

The BAL and BALR instructions store the whole of the P-counter in the return register. This includes the condition code, which may be stored by the following instructions, and tested later.

Name	Operation	Operands	Comments	Significant spaces indicated thus: ⌴ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	BALR	1,0			
	ST	1,MARCEST			

Check digit calculation and verification usually involves multiplication of the digits of the number by their weights. This can be avoided by translating the numbers to their pre-multiplied equivalents (which are stored as displacements from the modulus, to keep the number small).

Example

A check digit is to be calculated for the number 5215, using weights 1, 2, 3, 4, and modulo 11.

F5	F2	F1	F5	
	↓			NC against mask, giving digit position in zone quartet.
35	22	11	05	
	↓			Translate against pre-multiplied table.
09	06	02	05	

These numbers are then added together and the total divided by 11. This gives the check digit.

4–30 coding techniques

CLEARING

The easiest way of clearing a register is to load it with a zero address.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ⌴ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	LA	4,0			

To clear an area of core (maximum 256 bytes) an exclusive OR can be performed on that area.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ⌴ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	XC	STORE1,STORE1			

TESTING

In order to test the state of a register the condition code can be set by adding to it a half-word containing zeros.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	AH	4,HLFWORD			

To test an area of core for zero or non-zero, the condition code can be set by performing a logical OR on that area.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	OC	STORE1,STORE1			

To test a single byte in core for either zeros, ones or both, the test under mask of all bits should be used.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	TM	BYTE1,X'FF'			

To test a packed decimal field (having standard sign quartet of either C, D or F) for positive or negative only, the condition code may be set by testing the last byte of the field under a mask of bits 6 and 7.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	TM	BYTEX,X'03'			

The field is negative if the selected bits are mixed.

EXCHANGING

If the contents of two areas of core (of equal length but not exceeding 256 bytes) are to be exchanged without the use of a third area of core, then three exclusive OR instructions should be used.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	XC	STORE1,STORE2			
	XC	STORE2,STORE1			
	XC	STORE1,STORE2			

If the two halves of a single byte are to be exchanged, it is only necessary to use the PACK (or UNPK) instruction.

Example

Name	Operation	Operands	Comments	Identification
	PACK	BYTE1,BYTE1		

ALTERING BRANCH INSTRUCTIONS AND SWITCHING

If the conditional branch is to be altered to a NOP instruction, then a logical AND should be used on the second byte of the instruction with a mask of bits 4 to 7.

Example

MASK1 contains X'0F', then:

Name	Operation	Operands	Comments	Identification
	NC	BRANCH+1(1),MASK1		

If a NOP instruction is to be altered to an unconditional branch instruction, use a logical OR on the second byte of the instruction with a mask of bits 0 to 3.

Example

MASK2 contains X'F0', then:

Name	Operation	Operands	Comments	Identification
	OC	NOBRANCH+1(1),MASK2		

Subroutines may be entered at different points by use of the BAL instruction, which allows multiple switching.

Example

Calling module

Name	Operation	Operands	Comments	Identification
	L	15,=V(SA)		
	BAL	14,16(0,15)		

Subroutine

```
Name     Operation Operands                Comments
         USING     ZA,15
ZA       B         LFA
         B         LFB
         B         LFC
         B         LFD
         B         LFE
```

Multi-way switches are best handled using S-type constants.

Example

A switch is to be made on the run-type which is held in a half-word binary location WAARNTP. Register 5, the base register for S-type constants, is used here as an index register.

```
Name     Operation Operands                Comments
         L         5,=A(SAASWTAB)          A(S-CONSTANTS) FOR SWITCH.
         AH        5,WAARNTP               TWICE BECAUSE S-TYPE CONSTANTS
         AH        5,WAARNTP               2 BYTES LONG
         MVC       LB1+2(2),0(5)           S-CONSTANT TO BRANCH INSTRUCTION
LB1      B         LA                      LA IS DUMMY ADDRESS

         USING     SAASWTAB,5
SAASWTAB DC        S(LC)                   FOR RUN-TYPE 0
         DC        S(LE)                   FOR RUN-TYPE 1
         DC        S(LG)                   FOR RUN-TYPE 2
         DC        S(LI)                   FOR RUN-TYPE 3
         DC        S(LK)                   FOR RUN-TYPE 4
         DROP      5
```

ROTATION AND SHIFTING

It should be emphasized that the MVC instruction may be used to shift an area of core only to the left as the bytes are moved starting with the left-most byte of the area. An attempt to shift to the right with this instruction would cause the original source field to be corrupted before the instruction has been completed.

The MVC can be used to repeat a field in store.

Examples

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 Significant spaces indicated thus: ␣ (1 space) 71	73 80
	MVC	AREA+1(9),AREA		

This will cause the contents of the byte AREA to be repeated in the 9 consecutive bytes starting at AREA+1.

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 Significant spaces indicated thus: ␣ (1 space) 71	73 80
	MVC	AREA+2(8),AREA		

This will cause the contents of the two bytes AREA, AREA+1 to be repeated in the four consecutive two byte locations starting at AREA+2, AREA+4 and so on.

INDEXING

If an index is to be loaded with a positive constant in the range 0 to 4095, the load address instruction with a self defining value should be used.

Example

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 Significant spaces indicated thus: ␣ (1 space) 71	73 80
	LA	4,SDV		

To increment an index with a positive constant in the range 1 to 4095 use the load address instruction together with a displacement equal to the increment required.

Example

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 Significant spaces indicated thus: ␣ (1 space) 71	73 80
	LA	4,SDV(4)		

To place the sum of two registers in a third register, the load address instruction should be used. A small positive constant may also be added at the same time, in the range 0 to 4095 as shown in the above example. The following options are available.

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 Significant spaces indicated thus: ␣ (1 space) 71	73 80
	LA	4,0(5,6)		

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	LA	4,SDV(5,6)			

Care should be taken when using the load half-word instruction (LH) to load an index, as the most significant bit is taken as the sign and generated through the left half of the register.

There is no real difference between a base register and an index register. If care is exercised, base register modification can effect the indexing of SS instructions or can provide double-indexing for RX instructions.

BINARY ARITHMETIC

When concerned with only the least significant 24 bits (or less), any of the indexing techniques may be used.

When concerned with only the least significant 12 bits (or less), the load address instruction may be used to subtract small positive constants in the range 1 to 4095.

Example

The amount 735 is to be subtracted from the least significant 12 bits of register 4.

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	LA	4,4096-735(4)			

It should be noted that a carry into the 13th bit (bit 19) will occur, so that no more than 12 bits of the result can be used.

In order to conserve space and time, half-word constants and instructions should be used, taking care to remember that the most significant bit is taken as the sign.

CHARACTER MANIPULATION

Care should be taken when using the load half-word instruction on a 2 byte character field in a register remembering that the most significant bit is taken as the sign.

The load address instruction can be used to load a 1-byte character into a register.

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	LA	4,C'+'			

An area of core may be filled with a particular character by using the following sequence of instructions:

Example

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40 71		73 80
	MVC	STORE1,CHAR			
	MVC	STORE1+1(L'STORE1-1),STORE1			

MISCELLANEOUS

The BAL and BALR instructions store the whole of the P-counter in the return register. This includes the condition code, which may be stored by the following instructions, and tested later.

Name 1 8	Operation 10 14	Operands 16	Comments Significant spaces indicated thus: ␣ (1 space) 40 71	Identification ' 73 80
	BALR	1,0		
	ST	1,WAACCST		

Check digit calculation and verification usually involves multiplication of the digits of the number by their weights. This can be avoided by translating the numbers to their pre-multiplied equivalents (which are stored as displacements from the modulus, to keep the numbers small).

Example

A check digit is to be calculated for the number 5215, using weights 1, 2, 3, 4 and modulo 11.

F5	F2	F1	F5	
	↓			NC against mask, giving digit position in zone quartet.
35	22	11	05	
	↓			Translate against pre-multiplied table.
09	06	02	05	

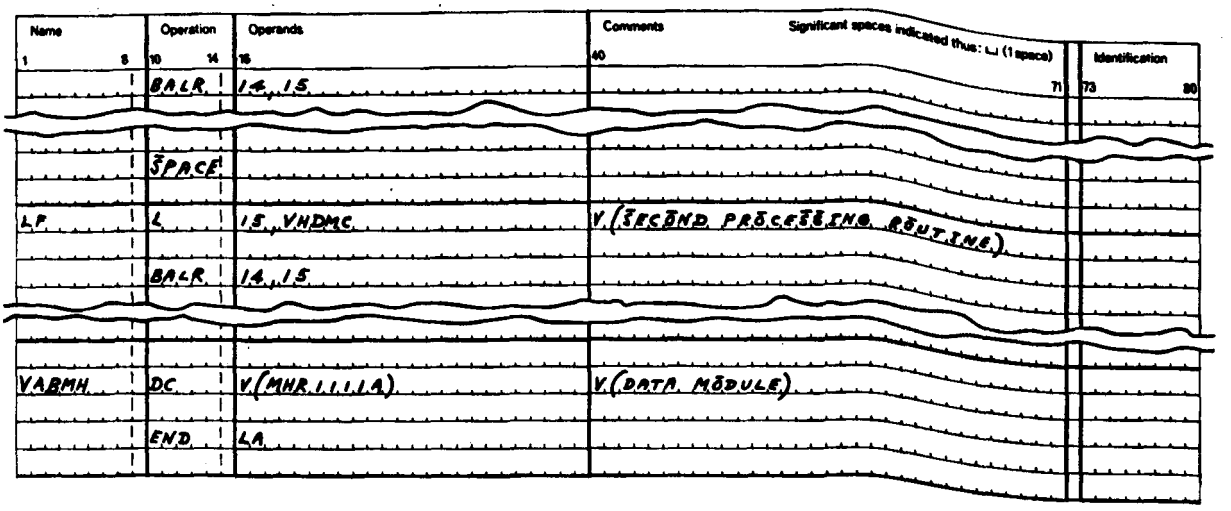
These numbers are then added together and the total divided by 11. This gives the check digit.

EXAMPLE OF USE OF STANDARDS

This example does not purport to be good Usercode; it is not, in fact, a complete program, but merely sections of one, showing the working of the standards in a complicated set-up.

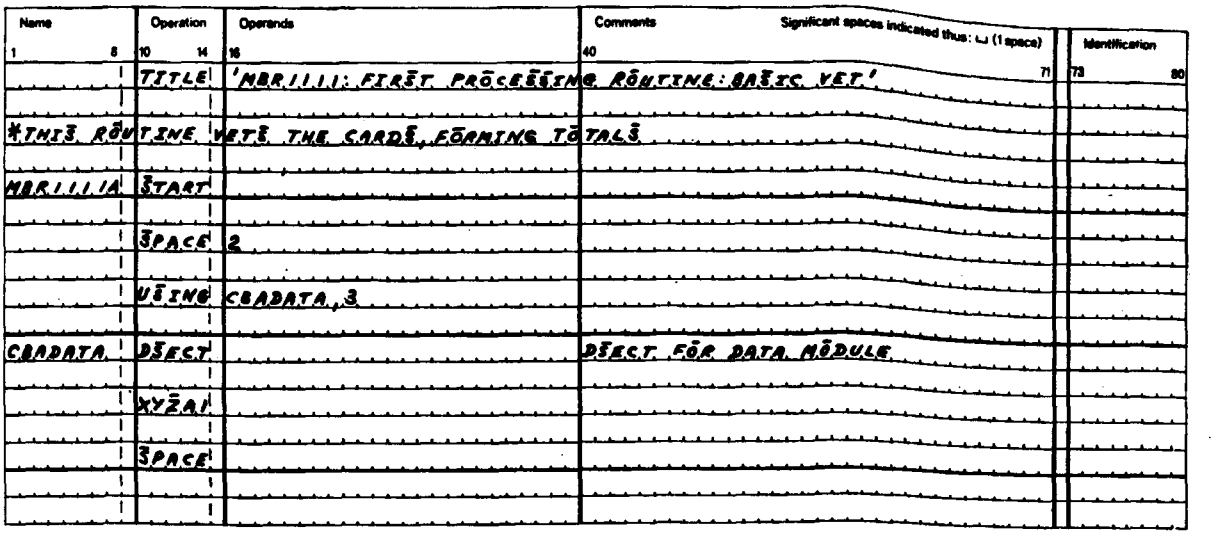
Name 1 8	Operation 10 14	Operands 16	Comments Significant spaces indicated thus: ␣ (1 space) 40 71	Identification 73 80
	TITLE	'XYZR1111:VETTING ROUTINE'		
MAR1111A	START			
	SPACE	2		
	USING	LA+2,4		
LA	BALR	4,0		
	L	3,VABMH	V(DATA MODULE)	
	USING	CADATA,3		
CADATA	DSECT		DSECT FOR DATA MODULE	
	XYZAI			
MAR1111A	CSECT			
	LA	13,WNFWORK	DYNAMIC WORK AREA	
	SPACE			
LC	L	15,VHDMR	V(FIRST PROCESSING ROUTINE)	

Note: the V-constant is defined in the data module, hence the module code H

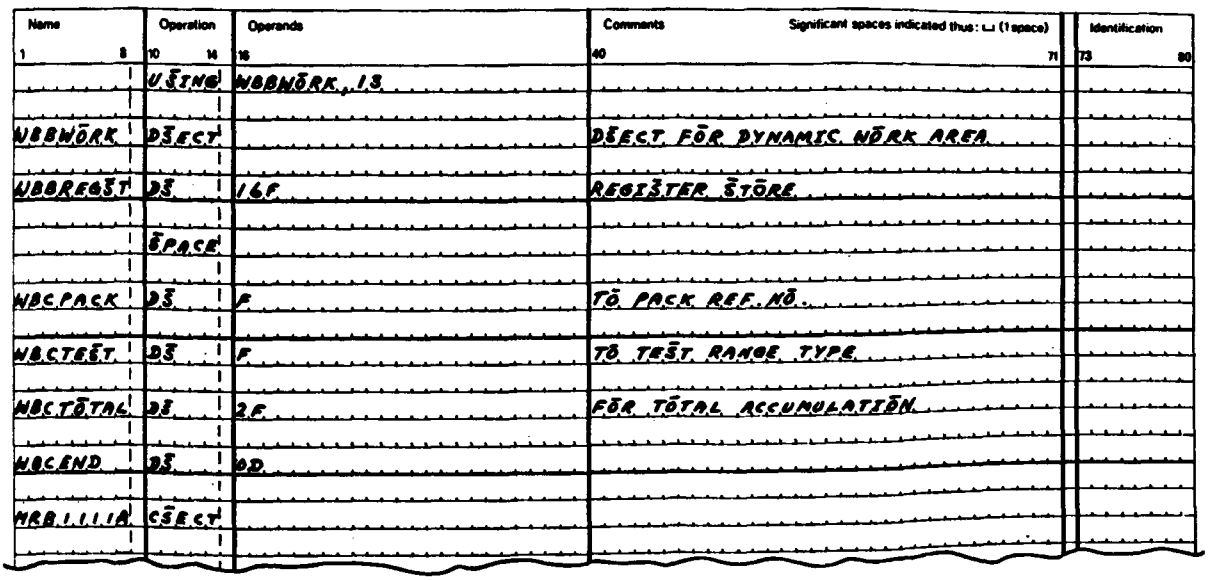


Name	Operation	Operands	Comments (Significant spaces indicated thus: ␣ (1 space))	Identification
	BALR	14,15		
	SPACE			
LP	L	15,VHDMC	V(SECOND PROCESSING ROUTINE)	
	BALR	14,15		
VABMH	DC	V(MHR1111A)	V(DATA MODULE)	
	END	LA		

The first processing routine might be:



Name	Operation	Operands	Comments (Significant spaces indicated thus: ␣ (1 space))	Identification
	TITLE	'MHR1111: FIRST PROCESSING ROUTINE: BASIC VET'		
*THIS ROUTINE	VETS THE CARDS, FORMING TOTALS			
MHR1111A	START			
	SPACE	2		
	USING	CBADATA,3		
CBADATA	DSECT		DSECT FOR DATA MODULE	
	XYZAT			
	SPACE			



Name	Operation	Operands	Comments (Significant spaces indicated thus: ␣ (1 space))	Identification
	USING	WBBWORK,13		
WBBWORK	DSECT		DSECT FOR DYNAMIC WORK AREA	
WBBREGST	DS	16F	REGISTER STORE	
	SPACE			
WBCPACK	DS	F	TO PACK REF. NO.	
WBCTEST	DS	F	TO TEST RANGE TYPE	
WBCTOTAL	DS	2F	FOR TOTAL ACCUMULATION	
WBCEND	DS	0D		
MHR1111A	CSECT			

```
                SPACE
LGA             L       15,YHDEB        S/R TO VET ALPHA CHS.
                LA      13,WBCEND       START OF FREE WORK AREA
                LA      1,4(0,5)        START OF FIELD TO BE VETTED
                BALR    14,15
                SH      13,YHCB         RESTORE R13.
                SPACE
```

The first subroutine module might start:

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 Significant spaces indicated thus: ␣ (1 space) 71	73 80
	TITLE	'SUBROUTINE MODULE: S/R ZA'		
MDR1111A	START			
	SPACE			
	USING	CDADATA,3		
CDADATA	DSECT		DSECT FOR DATA MODULE	
	XYZA1			
	SPACE			
	USING	WDBWORK,13		
WDBWORK	DSECT		DSECT FOR DYNAMIC WORK AREA	

Name	Operation	Operands	Comments	Identification
1 8	10 14	16	40 Significant spaces indicated thus: ␣ (1 space) 71	73 80
WDBREGST	DS	16F	REGISTER STORE	
MDR1111A	CSECT			
	SPACE	2		
*S/R ZA	DATE VE	T: STORES CC IN ONLY PARAMETER		
*CC SET	AT 1 IF	DATE O.K.		
	USING	ZA,4		
ZA	STM	0,15,WDBREGST		
	LR	4,15		

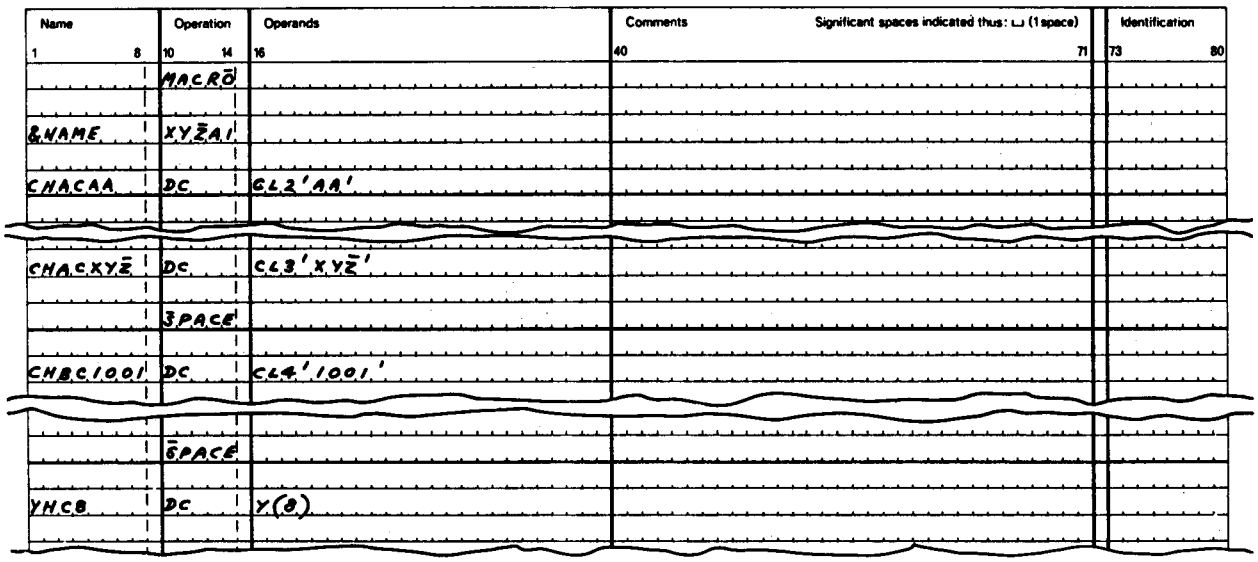
```
                LM      0,15,WDBREGIST
                BR      14
                SPACE
NDCMVC          MVC     WDBSTORE(4),0(1)        TO MOVE VARIOUS PARTS OF DATE
                SPACE
SDDLGCB         DC      S(LGCB)                 S-CONSTANTS FOR SWITCHES
SDDLGCC         DC      S(LGCC)
SDDLGCD         DC      S(LGCD)
SDDLGCE         DC      S(LGCE)
```

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	TITLE	'SUBROUTINE MODULE:S/R ZB'			
	SPACE	2			
*S/R ZB V	ETS ALPHA FIELDS: PUTS IN * IF		FIELD NOT O.K.		
*FIELD TO	BE VETTED IN R1				
	USING	ZB,4			

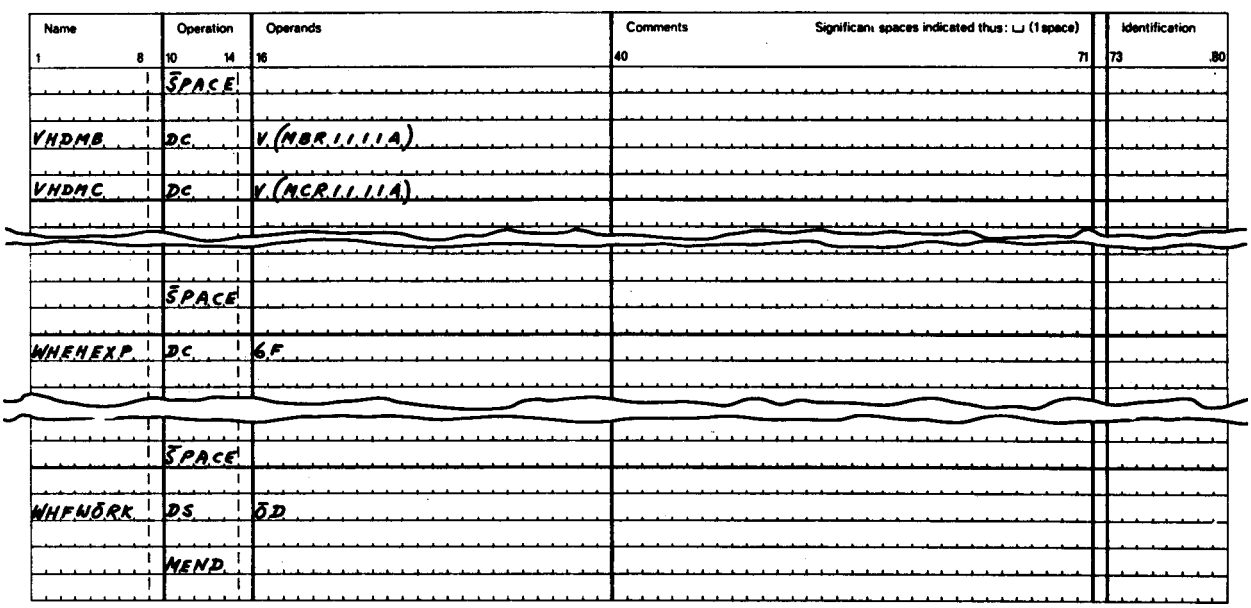
Finally, the data module might be:

Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	TITLE	'MHR1111:DATA MODULE'			
MHR1111A	START				
	USING	CHADATA,8			
	CNOP	0,8			
CHADATA	XYZA1				
	END				

And the macro XYZA1 might be:



Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	MACRO				
&NAME	XYZA1				
CHACAA	DC	C'2''AA''			
CHACXYZ	DC	C'3''XYZ''			
	SPACE				
CHBC1001	DC	C'4''1001''			
	SPACE				
YHCB	DC	Y(0)			



Name	Operation	Operands	Comments	Significant spaces indicated thus: ␣ (1 space)	Identification
1 8	10 14	16	40	71	73 80
	SPACE				
VHDMB	DC	V(MBR1,1,1,1A)			
VHDMC	DC	V(MCR1,1,1,1A)			
	SPACE				
WHEHEXP	DC	6F			
	SPACE				
WHFWORK	DS	0D			
	MEND				

Appendix 15 COBOL coding standards and techniques

The aim of this appendix is to assist programmers in making the most efficient use of the COBOL language and to guide them in laying out a program in a neat, readable and, where possible, standard manner.

There are three sections. The first is a general section on standards and techniques which apply to both 1900 Series and System 4 machines. The second and third sections are concerned with specific techniques related to the 1900 Series and System 4 machines respectively.

Although being primarily concerned with the full COBOL language these standards and techniques apply equally to COMPACT COBOL wherever the relevant facilities exist.

PROGRAMMING STANDARDS AND TECHNIQUES

Layout

The file definitions and known working storage areas of a program should always be written before starting the Procedure Division. A well-defined Data Division simplifies the actual coding and is essential to the object efficiency of the program. The Identification and Environment Divisions may be written after the other Divisions have been completed.

Standard coding sheets should be used and blank lines should be left between statements so that amendments may be inserted without rewriting existing coding.

The following should start on a new coding sheet:

Each of the four divisions

Each file description

Each section in the Data or Procedure Division

Individual record descriptions and paragraphs should only continue from one coding sheet to another if this cannot be avoided, that is, when they are too large for one sheet.

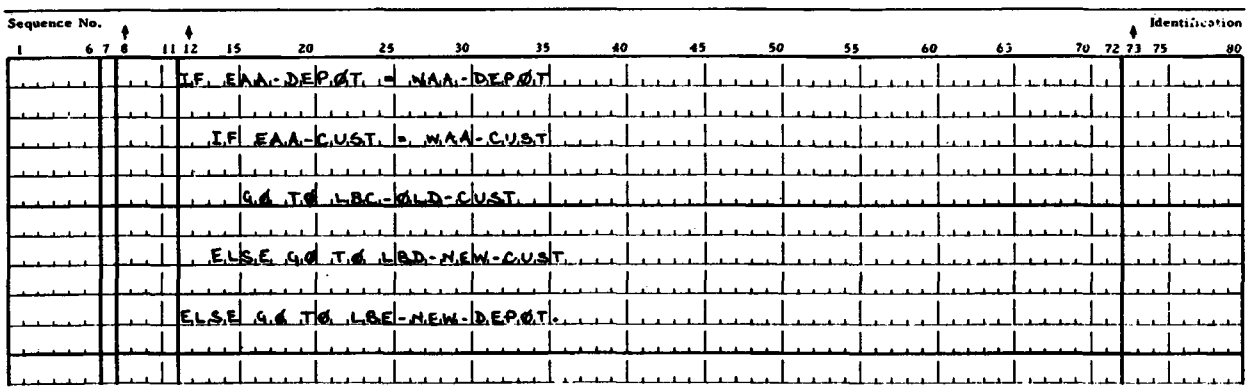
Data names and level numbers should be indented a number of characters, according to their level, so that groups of items are readily identifiable. The PICTURE and USAGE entries should be aligned to the same column for each elementary item.

A paragraph name should be written on a line without other coding. This makes each paragraph more easy to recognize, and helps in following and amending the logic of a program.

Each verb should form a new line of coding, that is, there should be only one instruction to a line.

Paragraph names should always start on column 8. Most statements in a paragraph should commence on column 12; however, it is helpful if continuation lines are indented.

IF/ELSE statements can be made easier to follow if the following system of indentation is used:



The ELSE option is not available to COMPACT COBOL but the concept of indentation is still valid.

Full-stops and quotation marks are the only essential punctuation characters, and all others should be avoided since they are likely to cause data preparation errors.

Full-stops are important in COBOL and any omission can cause program errors. All statements, with the exception of those included in conditional sentences, should therefore end with a full-stop. Although this is not strictly necessary it is a safer practice. If full-stops are present less errors are likely to occur if statements are moved from one part of a program to another, particularly when they are moved to the end of a paragraph. In addition there are some cases when the compiler moves on to the next full-stop, rather than the next verb, if a confusing error occurs in the sentence.

Referencing standards

DATA NAMES

These standards attempt to fulfil the following requirements of good data names:

1 They should be intelligible to any programmer

2 They should be systematically locatable within the module and program

3 A meaningful data name should be possible

4 The previous aims should be achieved with a minimum overhead of non-readable coding

The data name is constructed of a three character code, a hyphen and a free form data description. These are used as follows:

1st character	File code: This is the letter assigned to the file throughout the program, and may be in the range B, D to H, J, K, P, Q, R, T, U, X. Working storage: The letter assigned is W for internal storage and work areas and C for constants. The other letters have special significance in other languages and are omitted for compatibility.
2nd character	This is an alphanumeric code for the module or independently compiled routine, and is in the range A to Z, 1 to 9. The letter A should be for the control module, or for the program if it is a single module program. Data described in the Linkage Section should take the code of the module in which it is originally defined.
3rd character	This is an alphanumeric code for the record within the file or the group within the working storage, and is in the range A to Z, 1 to 9.
4th character	A hyphen is used to separate the prefix from the meaningful data description.
Meaningful data name	This is a free form description of the data. As an inbuilt uniqueness is inherent in the reference system it should be possible to use the same name for equivalent data in different files. *Example* MOVE EAA-QUANTITY TO HAB-QUANTITY. This could be used to move a quantity from the input record to the output record.

SOURCE LIBRARY EXTENSIONS

A degree of predefinition of the Data Division may at times be useful. Two areas may be predefined.

1 Files in a suite of programs should be given unique alpha codes (non-standard letters may be used if there are more than 14, or files required by one program only may be given common codes according to their use) and defined in COBOL terms. They may then, where the compiler allows it, be put on a library for the use of several programs

2 Date required for module linkage should be predefined and where possible put on the library

One advantage of this method is that if several modules need to be merged into one, the module calls can be replaced by PERFORMS and the Linkage Section statements removed.

This predefinition will save time as the information need only be coded in full once, and can be incorporated into each program or module by use of the COPY statement. It will also ensure that no misunderstanding can occur and that standard descriptions are used throughout.

PARAGRAPH NAMES

Paragraph names should conform to the labels of the program flowchart, that is, the flowchart box reference prefixed by L followed by a hyphen and a meaningful paragraph name. For ease of reference paragraphs should be ordered alphabetically within a program.

Although on some machines there is a small storage penalty for using paragraphs they should be used freely to show the logic of a program. Very long paragraphs tend to make the program more difficult to follow.

File definition

All references to files should, as far as possible, be in the same order. Files should be assigned in order of their alpha code and they should be defined in the same sequence. Records should be defined in the order they are mentioned in the DATA RECORDS ARE clause.

The File Definition should show as much useful information as possible and each clause should start on a new line.

The most frequently referenced files should be defined first. On 1900 Series machines they are less likely to be forced into Upper Variable areas and on System 4 computers they will be assigned prime use of the available base registers.

Data definition

Where possible 77-levels should be avoided as these do not normally help in following the program. Working storage items should be arranged in groups according to their data usage as only the group item need be defined as that usage, or they may be grouped logically. This also serves to order items according to their use and with regard to data name conventions, rather than to have them lost in a large section of 77-items.

88-level items should be used cautiously. Unless they are very thoroughly named they usually do not aid readability. 88-levels are not available in COMPACT COBOL.

PICTURE and USAGE entries should be aligned to the same column for each elementary item so they can easily be seen.

It is often useful to use only odd data levels so that new groupings may be made without altering the levels of subsidiary items.

No assumption should be made on the contents of working storage area at the start of a program since store is not necessarily cleared when a program is loaded. Particular care is needed when a program is being transferred from one version of COBOL which assumes clear store. It may be advantageous to initialize working storage entries using the VALUE clause to avoid any problems.

Arithmetic

The structure of the Data Division is very important with regard to the efficiency of processing numeric fields.

The following general rules should help to reduce unnecessary object coding to a minimum.

1 The Data Division should always be written before the Procedure Division. This will help to produce a logically more consistent and better planned Data Division

2 As far as possible usages of the data items involved in a statement should be the same

3 Decimal alignment should be the same. It is preferable to use some unnecessary storage in the Data Division as this will save storage in the Procedure Division

4 Numeric fields should always be signed, with the exception of 1900 Series Display decimal fields, whether or not the sign is required. Unsigned fields are considered as positive and may be adjusted to their absolute value every time they are used

5 Fields should be defined as alphanumeric (X) whenever there is a choice and are not required for arithmetic, even though they are specified as numeric. This will avoid unnecessary conversions

Subscripting

When data names are subscripted by variable subscripts the object code program calculates the address every time the item is accessed.

Double and treble subscripts are proportionately more expensive and the following technique should be used to reduce the number of variable subscripts required to a minimum.

1 Move the major subscript area to a work area

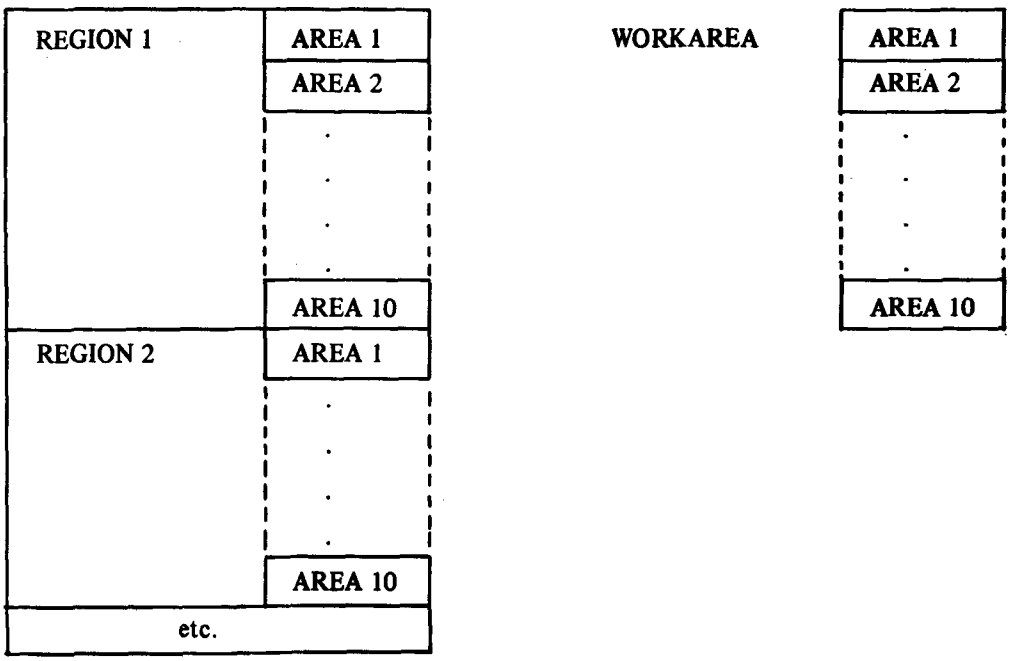
2 Perform all calculations on the work area using the minor subscript

3 Move the work area back to the major subscript area if necessary

Example

In this example REGION is the major subscript and AREA the minor subscript.

TABLE 1



The statements would then be:

```
Sequence No.                                                          Identification
1     6 7 8   11 12  15        20        25        30        35        40        45        50        55        60        65        70  72 73  75        80
      MOVE MAB-TABLE-1 (EAA-REGION) TO WAC-WORKAREA
      PERFORM ZA-CALCULATIONS
      NOTE  USE WAC-WORKAREA (EAA-AREA)
      MOVE WAC-WORKAREA TO MAB-TABLE-1 (EAA-REGION)
```

In some cases it is possible to access the work area without subscripting since each field of the work area could be defined separately.

Example

SALES 1	COST
	TIME
	CHARGE
	BONUS
SALES 2	COST
	TIME
	CHARGE
	BONUS
etc.	

WORKAREA

COST
TIME
CHARGE
BONUS

Similar techniques can be used to reduce the overheads on treble subscripting.

ALTER

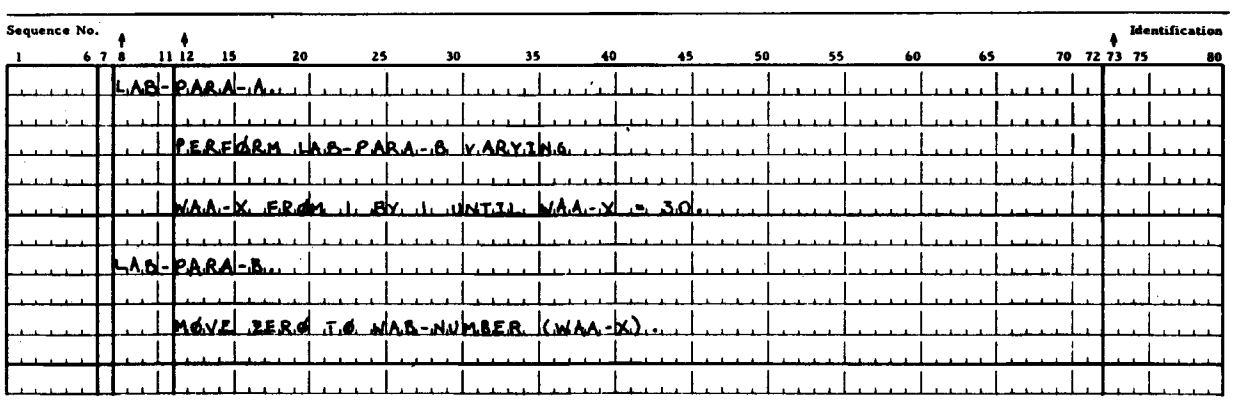
A GO TO statement should only be changed by the ALTER verb when it is absolutely essential, that is where the path of the logic changes irreversibly or when it is the only efficient solution. Such changes make the program more difficult to follow. When a GO TO statement is changed the ALTER statement should be coded as near to the GO TO as possible and a NOTE should be included.

Care should be taken when using ALTER statements on a segmented program. An ALTERed GO TO may be reset when the segment is next entered.

PERFORMS

In-line coding should not be subjected to a PERFORM statement. If a section of coding is required in more than one place it should be made a subroutine and handled as such. This may require an extra PERFORM statement but saves considerably in documentation and readability.

An exception to the rule for in-line PERFORMs is the PERFORM VARYING option where the following technique of looping is used:



The final area WAB-NUMBER (30) will be cleared after the PERFORM is completed.

The PERFORM VARYING facility is not available in COMPACT COBOL.

When a subroutine contains more than one paragraph it is quite useful to give it a section name. The PERFORM statement can then use this section name rather than the THRU option and paragraphs may be added to the subroutine without altering the PERFORM statement.

Conditionals

The use of extensively nested IF's should be avoided wherever possible since these are likely to cause confusion in checking and program maintenance.

When a number of tests have to be performed they should be ordered so that the most likely to be fulfilled occur first. This will reduce the number of tests to be made at run time before the correct action can be applied.

Nested IF statements are not available in COMPACT COBOL.

GO TO ... DEPENDING ON is a useful statement which is economical in both store usage and execution time.

Providing more than three tests are required to decide which action needs to be taken and there is not a great range of valid exits GO TO ... DEPENDING ON will require less store than the equivalent IF ... GO TO. Execution time will also be shorter as it involves address modification rather than a series of tests.

The statement GO TO ... DEPENDING ON has a built-in range test; items out of range cause control to be passed to the next statement, which could be used to check the range on input items.

The GO TO ... DEPENDING ON option is not available in COMPACT COBOL.

Annotation

Even though COBOL should be, to a great extent, self-documenting, certain annotation is essential. The following standards of annotation should help program documentation and maintenance.

1 The Identification Division should be fully coded. In particular the following details should be inserted.

  AUTHOR All programmers or organization concerned.

  REMARKS A brief description of the program.

2 The Data Division has no free form facilities for annotation but there are a number of optional comments and default options which may be used to aid readability.

  One non-standard method of annotation is to use the eight character program identity column (73–80). This is not recommended for, if programs are held on cards, the program name should be included in these columns to avoid confusion.

3 The Procedure Division should be adequately annotated using the NOTE statement. The following areas should always be annotated:

  Any ALTER statements

  Entries to and exits from other routines.

  NOTES may be emphasized by coding them on the right-hand side of the page. A NOTE statement is normally terminated by a full-stop, but if a NOTE is the first statement in a paragraph, the whole paragraph is considered as annotation.

1900 SERIES COBOL TECHNIQUES

The techniques included here have been written primarily to aid documentation and to reduce programming effort. Whenever possible economy of store and execution time have been considered and if further economy is required a re-appraisal of the use of COBOL rather than PLAN may be in order. On small machines data and procedure names may need to be shortened to get them into the compiler library.

Data definition and manipulation

All data should be defined using the PICTURE clause. CLASS, SIZE and SIGNED clauses should be avoided.

The CONSTANTS SECTION should not be used as it offers no advantages and is not compatible with other variations of COBOL.

The 77-levels should always be coded at the head of WORKING-STORAGE, for compatibility.

GROUPING ITEMS AND WORD ALIGNMENT

All items should be held on word boundaries wherever possible, that is all fields not required for external files and as many file entries as possible.

To ensure word alignment and to reduce the number of 77-level items, fields with the same usage should be grouped under 01-level entries. In this case it is only necessary to assign the usage to the group item. As 01-levels are automatically word-aligned elementary items are then described either as multiples of whole words or as SYNC RIGHT for numeric fields or SYNC LEFT for alphanumeric fields.

Thus COMPUTATIONAL fields are best described as S9(6) SYNC RIGHT or S9(12) SYNC RIGHT (this also avoids possible truncation problems), that is as single or double words.

Example

```
01  WAA-COUNTERS  COMP.
03  WAA-RECCOUNT  PICTURE S9(6)  SYNC RIGHT.
03  WAA-QUANTITY  PICTURE S9(12)  SYNC RIGHT.
03  WAA-LINECOUNT  PICTURE S99  SYNC RIGHT.
03  WAA-PAGECOUNT  PICTURE S9(6)  SYNC RIGHT.
```

If counters are held in this way it is possible to initialize them with one statement:

```
MOVE ZEROS TO WAA-COUNTERS.
```

It should be noted that the group item WAA-COUNTERS is itself an alphanumeric item and cannot be used in arithmetic operations.

MOVING DATA IN STORE

Word moves should be used whenever possible as these are more efficient than character moves.

Fields which have the usage COMP SYNC RIGHT will always be moved as words. Similarly fields where the relevant sending and receiving fields start and end on word boundaries and are the same length will be moved as words.

Word moves are generated where possible for both group and elementary fields; any odd characters are moved character by character.

Moves of small to large fields (requiring space filling) are always accomplished by character moves.

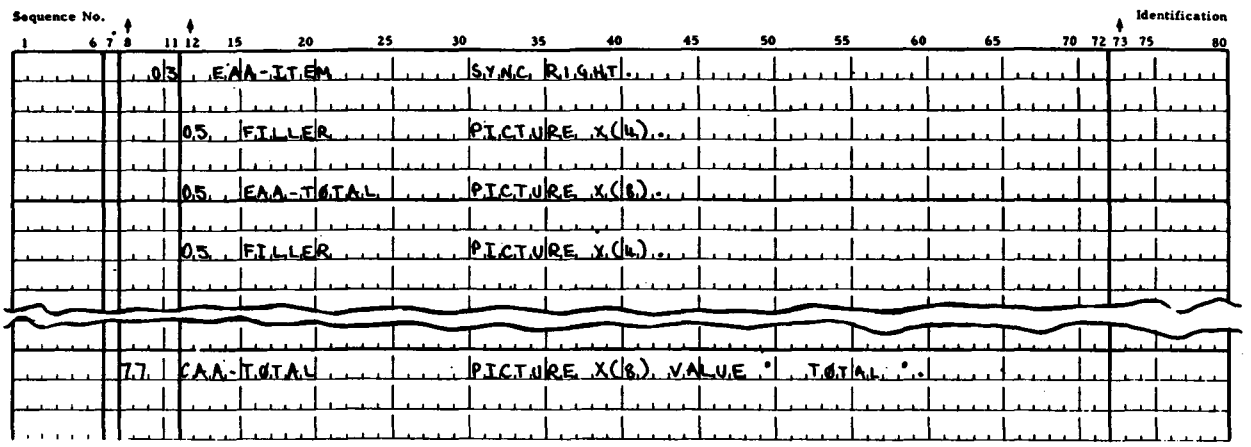
For compilation purposes the 1902A and 1903A source and object computers should be specified as 1904, in order to generate the most efficient moves.

The following is an example of how store layout may save space and time.

Part of a Print file is defined as follows:

```
03  EAA-ITEM  SYNC RIGHT.
05  FILLER  PICTURE X(6).
05  EAA-TOTAL  PICTURE X(5).
05  FILLER  PICTURE X(5).
77  CAA-TOTAL  PICTURE X(5) VALUE "TOTAL".
```

The statement MOVE CAA-TOTAL TO EAA-TOTAL requires 11 machine instructions. If the following coding is used no extra Data Division storage is required and only 4 instructions are needed. A considerable saving in execution time is achieved.



ARITHMETIC FIELDS

All fields in working storage used for arithmetic or subscripting should be defined as COMPUTATIONAL and be word-aligned. This is important as arithmetic performed on these fields (COMP SYNC RIGHT) usually requires no conversion and therefore generates fewer instructions than COMP-1 fields.

COMP-1 should only be used for compatibility with PLAN where mixed or decimal numbers are required. Fields in this format should always be SYNC RIGHT to ensure that they are held on a word boundary. Displays of COMP-1 fields will always be as bit patterns.

Any DISPLAY field on which arithmetic is performed more than once should previously be moved to a working location defined as COMP SYNC RIGHT (or word aligned). This results in only one binary conversion for a particular item in the object program.

SIGNED FIELDS

The conversion of display decimal fields to or from COMP or COMP-1 is more efficient if the fields are unsigned or at least the sending field is unsigned. Display decimal fields used for conversion or arithmetic should therefore only be signed where necessary.

Binary (COMP or COMP-1) fields however should be signed wherever possible as this sometimes saves instructions and reduces truncation problems.

Fields described as 1(24) or S1(23) will lead to double length arithmetic working while S1(22) fields will normally not.

EDITING

Editing generates a lot of coding as a generalized editing subroutine is not used in the object program. To economize on this the number of print line formats should be reduced as far as possible by aligning total-line fields with their detail-line equivalents.

If several fields are to be edited in the same way it is best to do this by subroutine. The field should be moved to a work area, from where it is edited in a subroutine, and the result then transferred as an alphanumeric string to the print area.

If a significant number of zero items are present, a branch around the editing may be in order.

VETTING

All numeric fields should be vetted on input from a slow device. It a non-numeric character is met during conversion it is ignored.

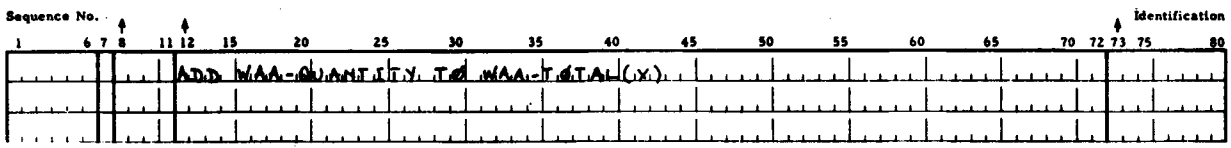
IF NUMERIC will accept leading spaces in an elementary numeric field as numeric but not in an alphanumeric field or a group item.

SUBSCRIPTING

Numeric items used for subscripting should be held in binary (COMPUTATIONAL) format. This also applies to counters used in PERFORM VARYING and GO TO ... DEPENDING ON statements.

When data names are subscripted by a variable subscript the object code recalculates the address every time it accesses the subscripted field, that is, it does not keep the calculated address even for use in the same statement.

Example



The object code will calculate the required address of WAA-TOTAL, add the contents of this address and WAA-QUANTITY in a work area, recalculate the address of WAA-TOTAL and store the result.

NON-NUMERIC LITERALS

A new non-numeric literal is created in the object program every time one is mentioned; therefore commonly used non-numeric literals should be set up as constants in working storage, and referenced by data-name. This does not apply to compilers XE1B or XEA3 which use common literal pools.

File handling

READ AND WRITE STATEMENTS

READ and WRITE statements generate up to 24 and 19 instructions respectively. Their number may be restricted by using PERFORM statements (4 instructions). It should be possible to limit the number of READ statements by the logical structure of the program, rather than by using PERFORMS. This will also serve to tighten control on the structure by giving the program fewer possible start points for each record processed.

At least one record defined in a print file must be the maximum length of the print line, that is 96, 120 or 160 characters depending on the printer.

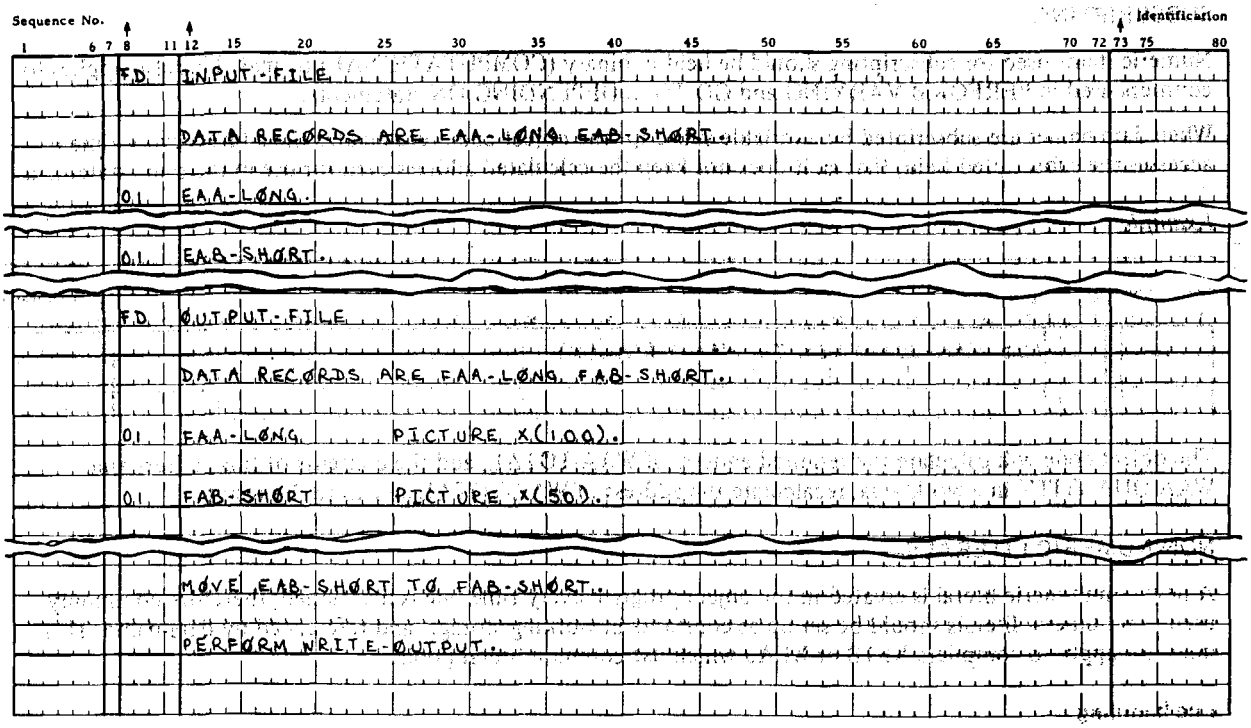
VARIABLE LENGTH RECORDS

Where input and output consist of a range of variable length records, care should be taken to avoid unnecessary moves. If a record is much shorter than the maximum record defined, it would be inefficient to let the compiler insert instructions to move the extra length from the input to the output area. Either the SAME RECORD AREA clause should be used to avoid the move, or separate records should be defined on the input and output files for each record length.

The SAME ... AREA option is not available in COMPACT COBOL.

Example

If an update file has two types of record, one 100 characters and one 50 characters long, the following techniques should be used for carrying across the shorter records.



This will both move and output 50 characters only.

If only one output record had been defined, FAA-LONG, and WRITE FAA-LONG FROM FAA-SHORT were used the whole 100 characters would be moved in store, although only 50 would be output since the output is controlled by the word count held at the start of the record.

SYSTEM 4 COBOL TECHNIQUES

The following techniques are based on coding in COBOL for the 4–50. They are of course also applicable to the 4–40 and 4–70, and to the 4–30 where the option concerned is available.

Data definition and manipulation

Great care should be taken in defining the Data Division, particularly with respect to data usages so that unnecessary conversions are avoided.

ARITHMETIC STATEMENTS

Calculations are likely to be carried out more efficiently if all fields concerned are defined with the same usage.

Numeric items should as far as possible be held in packed decimal format (COMPUTATIONAL-3) except for fields used for subscripting. The reduction in conversion time will compensate for any extra time required for calculations, especially as any division instruction or double word item will be processed in packed decimal. This may not be as significant in 4–70 as conversion and binary arithmetic are proportionately faster than packed decimal.

Care should also be taken with field sizes and decimal alignment of data used in arithmetic. Un-aligned fields will require extra instructions. Extra instructions are also sometimes required to truncate fields defined with an even number of digits.

Example

An ADD instruction is required on two packed decimal fields.

ADD A TO B		
PICTURE A	PICTURE B	Bytes
(COMPUTATIONAL-3)	(COMPUTATIONAL-3)	
S9(9)	S9(9)	6
S9(9)	S9(10)	10
S9(11)	S9(9)	18
S9(10)	S9(11)	22
S9(9)	S9(9)V99	40

MOVING AND COMPARING

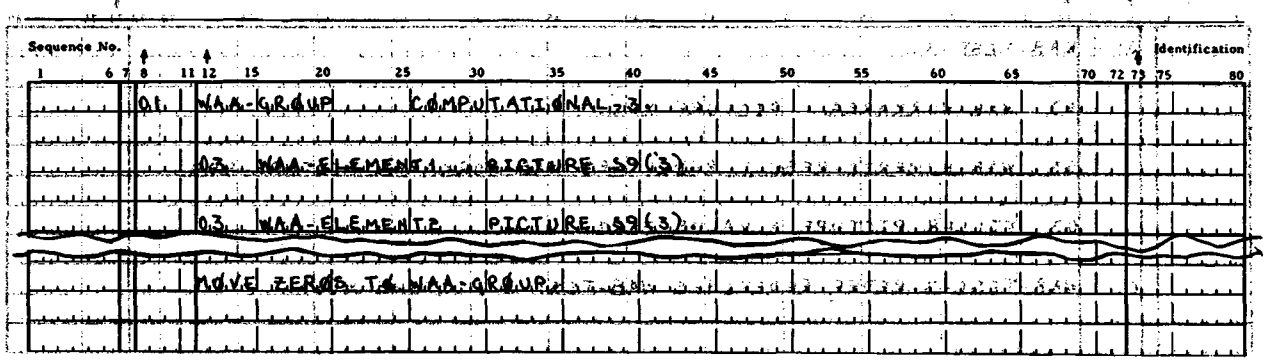
Care should be taken that the data descriptions of items which are to be moved or compared are compatible. The COBOL reference manual contains tables giving all allowable moves and compares, and doubtful cases should be checked.

It should be noted that on group or alphanumeric compares of different length fields, the comparison will take the length of the longest field and the shorter field will be extended to the right by spaces. External decimal fields are packed before being compared; thus no distinction is made between leading spaces and leading zeros.

When moving an external decimal field to a larger external decimal field, the compiler packs and unpacks the original number. This could be made use of, when dealing with punched cards, to remove overpunchings. Care should be taken in data vet programs that overpunchings which should cause data to be rejected are not lost before any error testing.

Group fields are always treated as alphanumeric and the usage is DISPLAY regardless of the definition of the elementary item.

Example



After this move each of the four bytes in WAA-GROUP will contain $(F0)_{16}$ rather than packed decimal zeros.

DATA VETTING

Vetting of numeric fields is extremely important as a System 4 program will fail to carry out an arithmetic operation on a field which is not numeric. A 5E program will abort and a 5J program will ignore the instruction and report the error. All data input from a slow device should be vetted but any data passed from one program to another on a magnetic storage device should be assumed to be correct.

When vetting numeric fields it is usually insufficient to use the IF NUMERIC test on its own. There are two reasons for this. Firstly it will reject an item with leading spaces and secondly it will accept an item with an overpunch on the least significant character.

If leading spaces are allowed, a statement EXAMINE *data name* REPLACING LEADING SPACES BY ZEROS should be used to change the spaces to zeros.

If overpunches are not expected, that is, when the field is unsigned, the field should be moved to another field which is one character longer and terminated by a zero. This field may then be tested.

Example

The field PICTURE XXX is moved to a work area WAB-WORKAREA which is part of the test field defined as follows:

```
Sequence No.                                                                    Identification
1     6 7 8  11 12  15   20   25   30   35   40   45   50   55   60   65   70 72 73 75   80
        01   EAA-HEADER.
        02   EAA-KEY COMPUTATIONAL-3.
        03   EAA-KEYA PICTURE 9(2).
        03   EAA-KEYB PICTURE 9(4).
        03   EAA-KEYC PICTURE 9(6).
```

Then if WAB-TEST-3-REDEF is tested and found to be numeric, the correct result is in WAB-NUMERIC.

Either a test field may be set up for each size of item to be tested, or, if leading spaces need to be catered for, a general area may be used with WAB-WORKAREA the size of the largest field required.

PACKED DECIMAL ITEMS

Care must be taken when defining packed decimal items so that the number of bytes required is calculated correctly. It is safest to define such items as an odd number of digits with a sign.

Example

```
Sequence No.                                                                    Identification
1     6 7 8  11 12  15   20   25   30   35   40   45   50   55   60   65   70 72 73 75   80
        01   WAB-TEST-3.
        03   WAB-WORKAREA PICTURE XXX.
        03   WAB-NUMERIC-RESULT REDEFINES WAB-WORKAREA PICTURE 999.
        03   FILLER PICTURE X VALUE ZERO.
        01   WAB-TEST-3-REDEF REDEFINES WAB-TEST-3 PICTURE X(4).
```

The three level 03 items require 2, 3 and 4 bytes respectively, and therefore the level 02 requires 9 bytes. If this 02 level were described in another record description as a FILLER, it would need the following definition:

```
Sequence No.                                                                    Identification
1     6 7 8  11 12  15   20   25   30   35   40   45   50   55   60   65   70 72 73 75   80
        02   FILLER PICTURE X(9) DISPLAY.
```

However it would be easy to make the mistake of adding the numeric digits 2, 4 and 6 and only allowing 6 bytes to cover the 12 digits. This error can be minimized if packed decimal fields are made up to an odd number of digits and signed. In the above example the fields would be S9(3), S9(5) and S9(7) which will give 9 bytes whichever way the calculation is carried out.

SUBSCRIPTS

Numeric items used for subscripting should be held in binary (COMPUTATIONAL) format. This also applies to counters used in PERFORM VARYING sentences. Store requirements do not increase greatly if other formats are used but execution time does.

When data names are subscripted by a variable subscript, the object code recalculates the address every time it accesses the subscripted field; that is, it does not keep the calculated address even for use in the same statement.

Example

For the statement ADD WAA-QUANTITY TO WAA-TOTAL (X) the object code will calculate the required address of WAA-TOTAL, add the contents of this address and WAA-QUANTITY in a work area, recalculate the address of WAA-TOTAL and store the result.

CORRESPONDING

The CORRESPONDING option should be avoided as it is difficult to establish which fields were intended to be affected when checking or maintaining the program.

Output files

COBOL builds up output records direct in the output area. Unless the program uses a work area, care must be taken that all fields are set up for each record. If constant fields are moved into the first record only, and the file is blocked and alternate areas are used, these fields will only appear on the first record of every other block.

When writing files in variable block format the APPLY WRITE ONLY option together with the WRITE FORM option gives a more economical use of magnetic tape. When the WRITE option is used the block is output when there is insufficient room for the largest specified record. When the WRITE FROM option is used the record is first moved to a compiler-reserved work area and the block is output only if there is insufficient space for this record in the output area.

4–30 COBOL restrictions

Some of the standards or techniques suggested are not suitable for 4–30 COBOL. The following is a list of these areas:

Data names and level numbers	In 4–30 COBOL the level numbers (01–10, 77) must start in margin A and the rest of the entry must not begin before margin B.
Subscripting	Only single variable subscripts are available in 4–30 COBOL. However if a two-dimensional table is required, double subscripting can be avoided by using the methods described.
	No binary usage is available in 4–30 COBOL. External decimal subscripts are shorter (requiring 26 bytes) than internal decimal (32 bytes).
	Addresses of variable-subscripted items are not recalculated in the same statement but stored and re-used.
External decimal moves	When moving one field defined as external decimal to another, the compiler does not pack and unpack the original field as in 4–50 COBOL. Instead, it inserts the source field into the right of the receiving field and pads the left of the receiving field with zeros.

Index

Amendment record

Should any additions or amendments to this Manual be necessary, they will be issued in the form of Amendment Lists.
When such changes have been incorporated, an appropriate entry should be made in the table below and the pink instruction sheet(s) should be filed behind this page.

List number	Item number
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to