G E N E R A L I Z E D E D I T I N G

(A DESIGN STUDY OF A COBOL ORIENTATED EDIT PROGRAM GENERATOR)

by Lance W Pearson B Sc

July 1978

A Thesis presented in partial fulfilment
of the requirements for the degree of
Master of Science in Computer Science
at Massey University

ABSTRACT

All commercial data processing installations include programs to detect errors in input data. There is a high degree of commonality in the editing (i e validating) of such input data throughout the data processing industry.

This thesis defines a generalized editing package which will allow a user to specify the editing requirements for any set of input data. From the specifications a COBOL program will be created to carry out the required operations on the input file.

Included as an introduction to this thesis, is a survey of editing needs, and a discussion on the merits of generalized software.

The thesis emphasizes the methodology of the generation of a specific "tailor-made" editor program.

Key Words: Editing
EPG (Edit Program Generator)
Generalized Software
Program Generator

Computing Review Category: 2.0, 3.50, 4.12, 4.41

ACKNOWLEDGEMENTS

I would like to express my deep appreciation to those people who have helped me throughout the writing of this thesis. It has been important to me to have their informative ideas and encouraging enthusiasm for this project.

Sincere thanks are due to my supervisors, Professor G Tate and Mr P J Melhuish, for all the able assistance and guidance they have given me. I particularly wish to thank Peter Melhuish for his many comments and suggestions offered after the careful reading of this thesis, even when it has been at his inconvenience. His personal friendship and advice I am grateful for.

I would like to take this opportunity to thank my parents for their consistent generosity and the optimism they have expressed to me throughout my years of study at Massey. The motivation this has been is appreciated and will not be forgotten.

Finally, I would like to convey thanks to my typist for her kind and competent assistance given during the preparation of this dissertation. She has made many sacrifices to share with me the headaches and nightmares that form part of the trauma associated with this type of excercise. ... A very special 'thank you' Karen.

Lance W Pearson
Massey University
Palmerston North
NEW ZEALAND

July 1978

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

1. INTRODUCTION - THE DP ENVIRONMENT

The processing of data, within a computer-assisted business typically follows a standardized set of procedures.

This is true, irrespective of the size of the organization or the number of systems requiring computer processing or the complexity of each procedural system.

Four areas common to data processing systems would be:

(1) Manual and clerical procedures

(2) Data capture, preparation, and input

(3) Computer processing and computer programming

(4) Report production and distribution

This is shown diagramatically in figure 1.1.

The diversity of manual and clerical procedures is great because the many industries have different services and objectives. This is not true for the other three common areas. They follow standard procedures which can be easily described.

In general the data processed is converted to some machine readable form. It is entered via an input device to be edited for errors and processed, then organized as a file structure held on some high capacity storage device. From it, readable documents conveying the input information in an organized and usable form, can be produced. The data to be processed can be very different in appearance. For example, the detail on items sold by a retailer barely resembles the detail on graded meat and its shipping destinations. Both are collections of transaction records. They can be processed in essentially the same way. For this reason the numerous models of computers used possess functional resemblances.

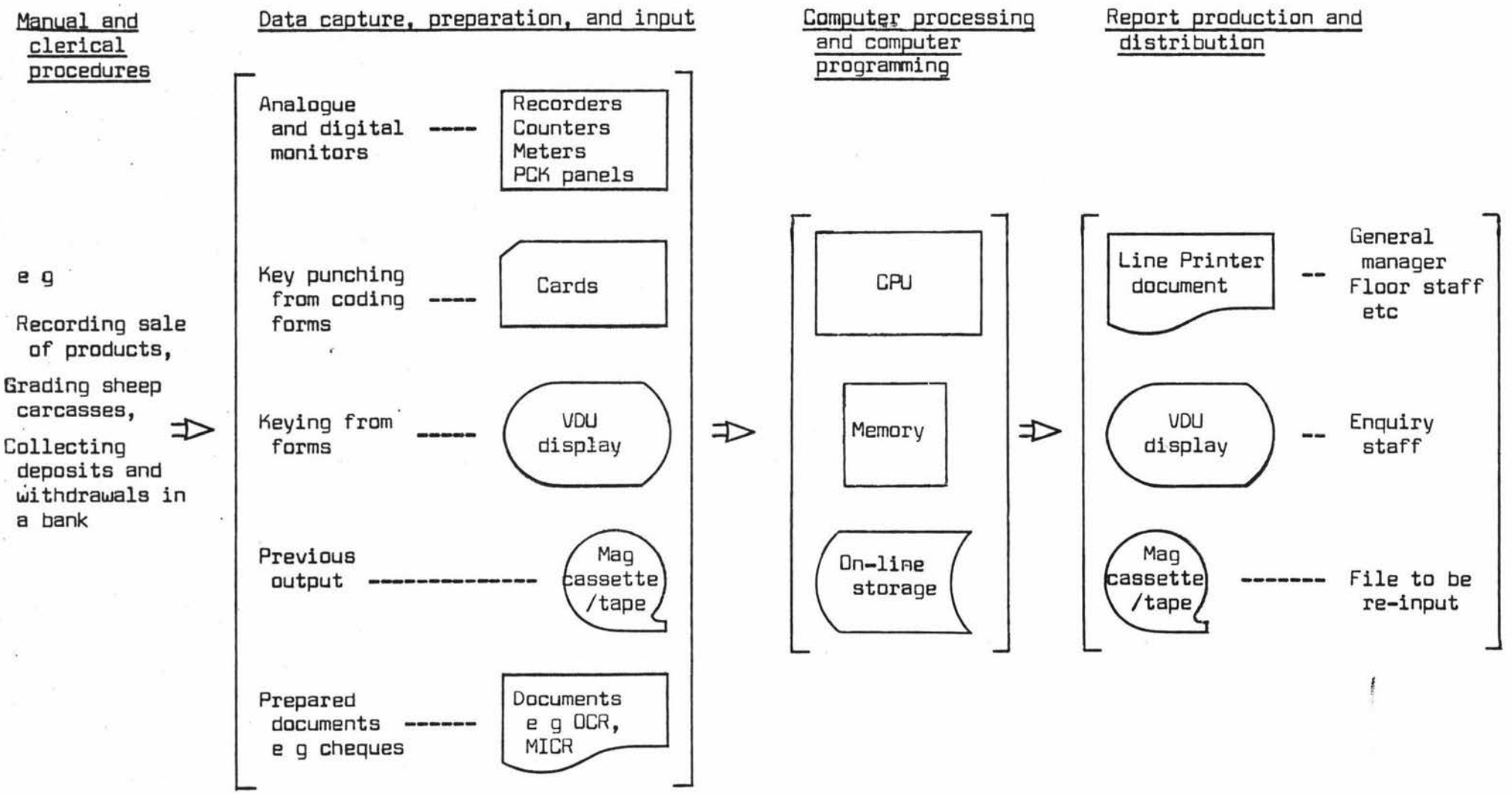


Fig 1.1 Common organization of data processing systems.

In data processing, user software aids exist to solve the many common problems needing computational processing. Program generators are such an example of generalized software aids. Error editing of input data is also an example of a common computational problem in this case shared by all computer users. This thesis is the specification for an edit program generator (EPG). Background discussion, on generalized software, the editing problem, and the alternatives to the solution of editing, precedes this specification.

2. GENERALIZED SOFTWARE

2.1 Use of Generalized Software

Within the computer industry much attention has been given to the development of generalized software. These are programs written to perform functions common to DP applications. They are generalized to allow modifications to suit the unique problems and needs of a wide range of possible users. In contrast, custom-built software is designed to solve the particular problems of one user. The latter is almost always less expensive to write and develop. However, generalized software is becoming more and more competitive in performance, efficiency and economy with tailor-made products because it is shared by more than one user.

In New Zealand most computer data processing is done on small scale machines for relatively small companies. There are a number of reasons why generalized software is attractive to them.

- Software development is time consuming (high level languages just aren't high level enough).
- Newly written in-house software is likely to be unreliable for some time.
- Generalized software to suit requirements is often available, from manufacturers, software houses, bureaux or other users.
- Software development principally requires labour. Labour is:-
  - expensive (more than the hardware costs)
  - unpredictable - length of time expected for programming
    - length of employment so that the software is adequately supported
  - often unavailable.
- Good documentation is often provided.

2.2 Examples of Generalized Software

Problem solution by the use of generalized software has evolved to increasing levels of generality as listed below:

(1) System software

(These appear at the lowest level of problem solution although they perform complex functions).

e g - Operating systems,
- Assemblers,
- Compilers,
- Message Control Systems,
- Management Information Systems,
- Data Base Management Systems.

(2) High level language extensions

e g - COBOL Sort and Report Writer,
- Indexed sequential access routines.

(3) Languages for specialized DP functions

e g - Reporters (such as RPG),
- On-line inquiry,
- Graphics packages,
- Text editing,
- Maths/Statistics functions.

(4) Applications programs

(These appear at the highest level of problem solution. They are usually independent enough so that data does not need pre or post processing).

e g - Payroll and labour cost packages,
- Inventory control.

2.3 Forms of Generalized Software

There are several forms of generalized software:

- code producing software,
- modified compilers allowing language extensions,
- macro- and pre-processors,
- parameter driven packages,
- program generator routines that will produce a program based on the specifications for a particular type of problem.

Each form has its advantages and disadvantages which will influence selection. Factors include:

- availability of software or programming resources,
- suitability for running on available hardware,
- degree of expertise required to use them,
- software support provided,
- type of application.

2.3.1 Code producing software

Like a compiler, code producing software generates unique machine-code from the user's specifications. This code is efficient but is machine dependent and therefore the generalized software can only be used on a subset of machines. Computer manufacturers are the primary source for such systems e g Burroughs DMS-II [1].

2.3.2 Extended compiler software, macro- and pre-processors

As with code producing generalized software, efficient code is generated but portability is a problem. Extended compilers do not promote standardization within the industry although some features gain enough acceptance to eventually become language features e g SORT verb in COBOL and more recently Indexed Sequential file

access routines, also in COBOL. These extensions never achieve total acceptance since some machines do not have adequate hardware to support the new features. In other cases the software enhancements required are prohibitively uneconomic.

2.3.3 Parameter driven packages

This software is completely generalized. As a result some degree of efficiency is lost. Each time a job is performed, all parameters have to be interpreted. As the job runs many data moves are necessary to communicate with the generalized procedures used by the package. The code for functions not being used occupies valuable space in the machine's memory. As a result the package is often too large for small to medium systems. The portability of parameter driven packages depends on the portability of the language used to write them.

2.3.4 Program generators

Program generators are the most flexible form of generalized software. They are as portable as the host language they are written in. The symbol-code produced by them can be just as portable, depending on the purpose this generalized software has. The programs they generate can be as compact and efficient as their custom-built equivalents. Their reliability and ease of use is immediate. Another popular advantage is that the symbol-code they generate is readily modified to suit user exceptions.

They have some disadvantages also. To produce a new program or modify an existing one the program generator must be run to generate symbol-code which then must be compiled into machine-code. This two-task operation is undesirable, particularly when a separate, larger machine must be used if the program generator is too large. This

also necessitates a third task of converting and transferring machine-code back to the application computer. Note, that this same problem may also be present with code producing software and extended compiler software.

Many RPG (Report Program Generator) implementations are program generators. Others are interpreters (i e Parameter driven packages). In New Zealand SPL, Systems and Programs (N Z) Ltd, have produced a program generator called "PROGENI" [2]. It is a collection of MACROS to assist COBOL programming.

3. THE EDITING PROBLEM

3.1 Certainty of Data Errors

The value and effectiveness of information produced by any data processing system depends on the accuracy and timeliness of the original data it uses. Human fallibility is the primary cause of data inaccuracies. Input data is the weakest link in the chain of data processing events. Therefore there is unanimous acceptance throughout the computer industry of: the inevitability of erroneous data, the likely effects they will have, the need to eliminate them, and the proportionate cost and effort this is expected to incur.

3.2 Causes of Data Errors

Knowing the causes of data errors is important if appropriate solutions are to be selected.

Errors can originate from inaccuracies in the source information or mistakes in the manual and clerical procedures of data capture, preparation, and handling. Occasionally they can also originate from computer hardware or software failures.

3.2.1 Inaccuracies in the source information

There are more types of source error than it is possible to identify. Some of the more common examples are:

- misspelling (e g Newzealand for New Zealand),
- omission (missing information),
- mixed detail (e g a name and address where the address belongs to someone else),
- untruths (plain wrong information).

3.2.2 Mistakes in the manual and clerical procedures

Errors possible in this area can arise at different procedural levels. Operational mistakes like incorrect naming of a file of transactions or delivering the wrong version of transactions to the data capture department both appear at the highest procedural level. At the transaction level forms can be missing or incorrectly ordered. The lowest procedural level is the activity of capturing individual transactions. Listed below are some mistakes that can occur at this stage:-

- Ill-constructed transactions (e g fields/groups of fields missing or in the wrong order. This can often occur when several source forms represent a single transaction, i e more than one physical record per logical record).
- Transposition errors (adjacent characters are swapped e g 1234 for 1324, or FINSD for FINDS).
- Substitution errors (wrong character keyed for various reasons as listed):-
  - a) Transcription error of easily confused characters e g (I-1), (Z-7-2), (Ø-O).
  - b) Shift error e g (U-1), (L-6), (Q-+). See Fig 3.1.
  - c) Adjacent key error strike by a common finger e g (A-S), (4-5), (4-7). See Fig 3.1.

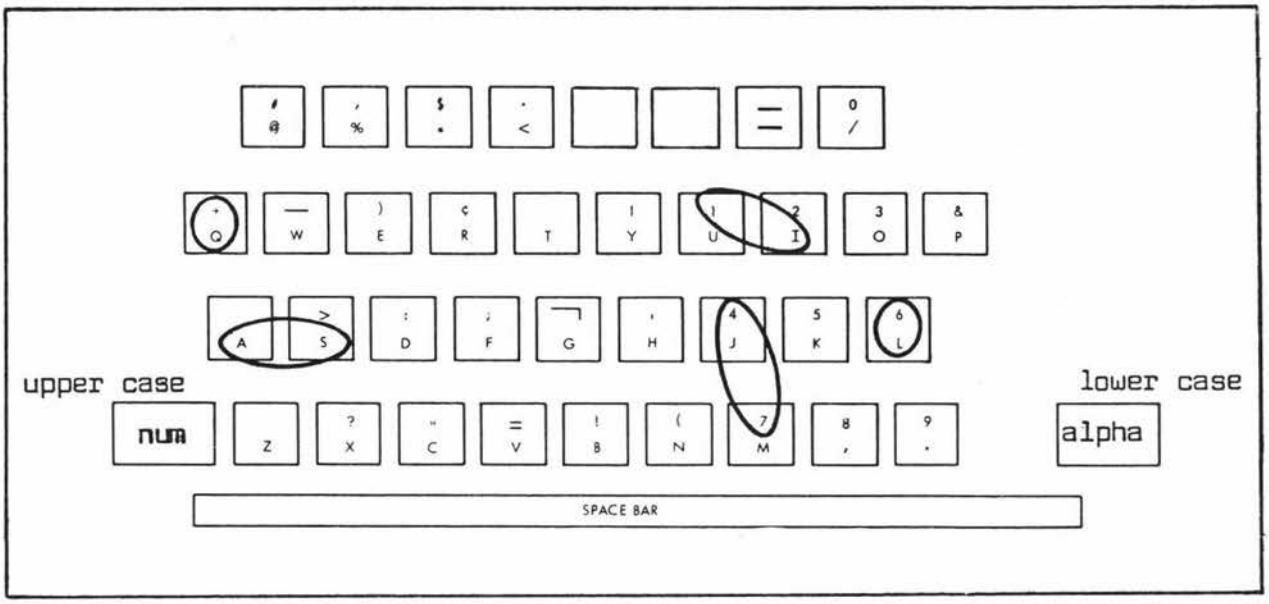


Fig 3.1 Standard keyboard layout showing lower and upper case characters, also the adjacency of characters.

Operator fatigue, operator inexperience, and illegible source documents are frequent causes of the manual and procedural mistakes discussed above.

3.2.3 Computer hardware failure

Data capture and processing devices are constructed to be part mechanical and part electronic. Data can be lost or changed as a result of either mechanical or electronic failure. Equipment can deteriorate because of mishandling or sabotage, power distrubances, normal wear, or environmental factors such as: temperature, humidity, dust, or in the worst case, natural disasters like fire, earthquakes, floods etc.

3.2.4 Computer software failure

As with hardware failures data can be changed or lost because of software malfunction. Failures like: bad address calculations, incorrect logic, or rounding and truncation side effects. Software failures may take a long time to discover and can reappear easily if program modifications are improperly made.

3.3 Effect of Errors

GIGO (Garbage In Garbage Out) is a well known term describing the effect of errors. The 'Garbage Out' can manifest itself as: incorrect or misleading reports, file corruption, loss of information, or malfunction of programs dependent for their working on the input data.

3.4 Cost of Errors

There are two costs which need to be distinguished. Firstly the cost of the effect of errors and secondly the cost of error prevention.

These costs are significant to all data processing organisations irrespective of their size or level of sophistication.

3.4.1 Cost of the effect of errors

If errors are permitted to pass through a data processing system then monetary costs, time costs, and asset costs result. For example, expensive reruns and file recreations may be necessary, there may be production delays, or customer/management dissatisfaction may arise if information is incorrect or too out of date to be of any use.

3.4.2 Cost of error prevention

Any error prevention measure such as: data verification, data validation/editing, and equipment safeguards, must be recognised as costs to an installation over and above the actual cost of processing the data. In an industry where 30 to 50 percent of the total working cost is incurred by data preparation, capture, and entry, error control is known to represent the highest overall individual processing cost [3]. This cost should be balanced against the importance of accuracy so that prevention-expense is less than cure-expense.

3.5 Techniques of Error Control

A great deal of manual, clerical and processing effort is directed at reducing the instances of errors in data. No control method is 100% reliable. The closer one attempts to attain 100% accuracy, the higher the cost incurred. Methods chosen are based on understanding the nature of errors. This includes a study of where they occur, how they occur, and what they look like when they do occur. Accepting the virtual certainty of input data inaccuracies

is an important start to solving this problem.

In parallel with section 3.2, on the causes of errors, the error control techniques discussed below can be grouped according to origin, i e techniques used: at the source, during manual and clerical procedures, and by hardware and software provisions.

3.5.1 Error control at source

There are several well established techniques used to improve the integrity of data right from its source. Reducing the number of times data must be recorded and copied is one method. The employment of well designed forms is another. This will not stop mistakes from being made. Other techniques are necessary to prevent the carrying of erroneous data throughout the processing. However an error is discovered, it is a useful improvement measure for the person who made it to be responsible for correcting it.

To detect loss or duplication of transactions a technique called batching is used. Here 25 to 50 transactions are grouped together as one unit. Selected fields of each transaction are used to produce control totals, hash totals, and document counts. (Discussed in chapter 5). These together with a count of the transactions in the batch are recorded on a transmittal-control route slip. (See Fig 3.2). Checking this information at the data capture stage not only helps to control the possibility of missing or duplicate records but also helps to find transcription errors in selected fields.

As part of the audit function computer generated reports, should be checked. These reports, often called 'audit trails', contain lists of input transactions together with messages relating to any instances of errors found. Often they also include comparisons between

computer calculated control totals, hash totals, etc, and those that were input with each batch of transactions. Strictly speaking an audit trial is the trail of a transaction right through a system. The edit reports are commonly the start of this trail.

Batch No.	To	
Date	From	
No. of documents	Numbered From	To
Control totals Hash totals		
Date rec'd	Rec'd by	

Fig 3.2 Batch route slip, to accompany transaction-source documents.

Important fields identifying the transaction, e g charge Account nos, employee pay nos, product nos, etc, can be assigned a check digit. (Discussed in chapter 5). This is also an effective measure to control transcription and mixed detail errors. Check digits are often a convenient by-product of encoding devices. Fields containing a check digit do not need to be verified. (Verifying is discussed in section 3.5.2).

An entirely different approach to information collection is to use methods not involving human intervention, for example bar-codes and magnetic-ink lettering. As yet these are expensive techniques with limited applications. Unfortunately they also do not provide an infallible data capture procedure, (e g damaged cheques may be misread).

3.5.2 Error control during manual and clerical procedures

The discussion in the previous section 3.5.1, on the use of batched transactions is also an important control technique relevant to this section.

At the most primitive level visual verification is used. This technique is slow and subject to the fallibility of the person involved. It should only be used if the information is to be processed immediately, as is the case in real-time systems of low volume data capture.

A more reliable method is to key verify. The devices used can read a keyed record and compare it character by character with a second keying operation of the same source document. New records can be produced if discrepancies are found. This method is only effective in preventing the key punching errors of transposition and substitution. It is expensive because double the effort and time is required for data conversion.

More intelligent data capture devices can greatly aid error control. If the device is able to perform check-digit calculations then those fields incorporating them do not need to be verified. Check-digit control is reputed to catch approcimately 97% of transposition and substitution errors [3]. This technique is normally used with numeric fields and may be expensive if the original check-digit has to be

hand calculated often. (See chapter 5 for check-digit examples).

The lowering of hardware costs has brought about the development of 'Audit Entry' equipment. These are intelligent data capture devices controlled by programmable mini-procesors. They have a keyboard, a printer, a disk or cassette drive, and sometimes a visual display unit, together with the processor, all built into the same unit. The Burroughs AE412 is a good example of modern audit entry equipment [4].

The use of Audit Entry equipment has become popular because of its extensive checking facilities. Not only can it perform verifying tasks and check-digit calculation checks but also it can check field ranges, and at a much higher level, do sequence checks and even control total checks on individual transactions or for entire batches.

Further alternatives to manual and clerical error control include the use of on-line data capture, specialized recording equipment, and optical document readers. On-line data capture requires the use of interactive editing software designed to perform all the error checking techniques of the standard off-line equipment. Checks on transaction completeness, sequence, and master file record correspondence are also possible using this method. Obviously computer time resources have to be available. Specialized recording equipment also requires software control, and hardware availability. This technique facilitates accurate recording by attaching sometimes complex event information to a single button depression or a controlled sequence of button depressions. Optical document readers at present (1978) have limited use. They are expensive and rely on a clearly written character subset.

3.5.3 Error control by hardware

The need for hardware controls to safeguard against loss or corruption of information has been recognized for many decades of computing now. Manufactures include enough control devices to ensure that hardware failures are rare. Devices such as: redundancy or parity bit error detection and correction, duplicate circuitry, echo checking, character code checking, multiple read heads on input devices, and read after write on I/O devices, are some examples. Maintaining environment controls like air conditioning, protected power supplies, regular service testing, and adequate security measures are important.

Hardware failures can and do occur despite the use of all safeguards mentioned. This possibility should be remembered. Often instances of uncontrolled hardware failure can be detected by manual or software checks. Some are virtually impossible to detect or control. For example, random and intermittent printer failures.

3.5.4 Error control by software

Software checking of data processed is the most powerful control technique. It can be used to detect mistakes made at the information's source, during the manual and clerical stage including data capture, by hardware failure, and sometimes by software failure itself. Listed below are some of the checks possible at the different levels shown:

- File level checks:
  correct file title,
  correct file version (date or serial no),
  transaction record and master file record correspondence,
  master file sequence.

- Batch level checks:

  control totals (e g hash totals, financial totals, document or record counts, and crossfootings).

- Transaction level checks:

  transaction construction (i e order and presence of physical records comprising logical records),

  copresence of dependent fields,

  sequence of transactions.

- Field level checks:

  presence,

  class (i e character subset e g numeric or alphabetic),

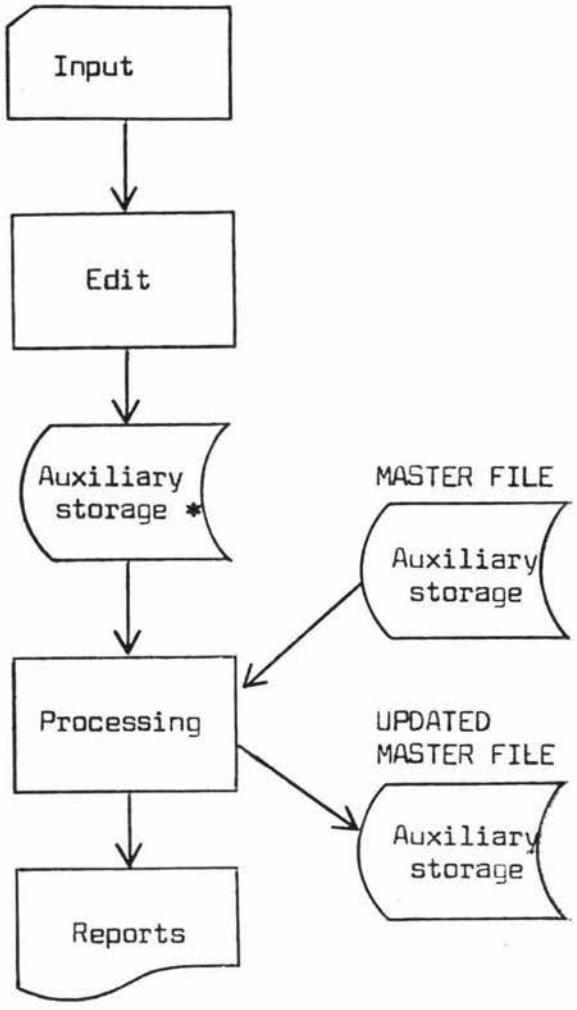
  range (i e limit or reasonableness values),

  check digit,

  table membership.

The use of software dedicated to performing these checks is standard practise. They are called editors or validators. (The most popular term is 'editor' and shall be used throughout this thesis). In a batch processing environment edit runs are performed after data entry preceding any intended processing of the information. (See Fig 3.3). Similarly, in a real-time processing environment, each transaction is edited before further processing is applied. (See Fig 3.4).

Actions taken when errors are detected vary between the batch and real-time processing situations. If batch processing is used erroneous transactions are typically listed in an error report together with some message indicating the type of error and what action has been taken e g zero or space over-writing of fields, rejecting the



*Note. This symbol is used
to represent both On-line & Off-line storage.

Fig 3.3 Edit precedes processing in the batch environment.

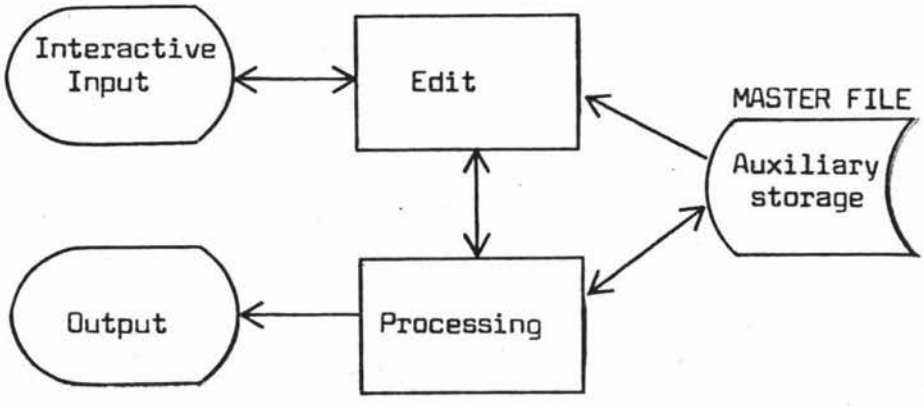


Fig 3.4 Edit precedes processing in the on-line environment.

transaction, or rejecting the batch. Rejected transactions and batches are often held as error files pending correction and re-editing. (See Fig 3.5). Immediate correction is possible in an on-line situation. Usually the operator/user is warned as soon as wrong fields of information have been entered. The correct field can then be entered or the transaction concerned can be skipped.

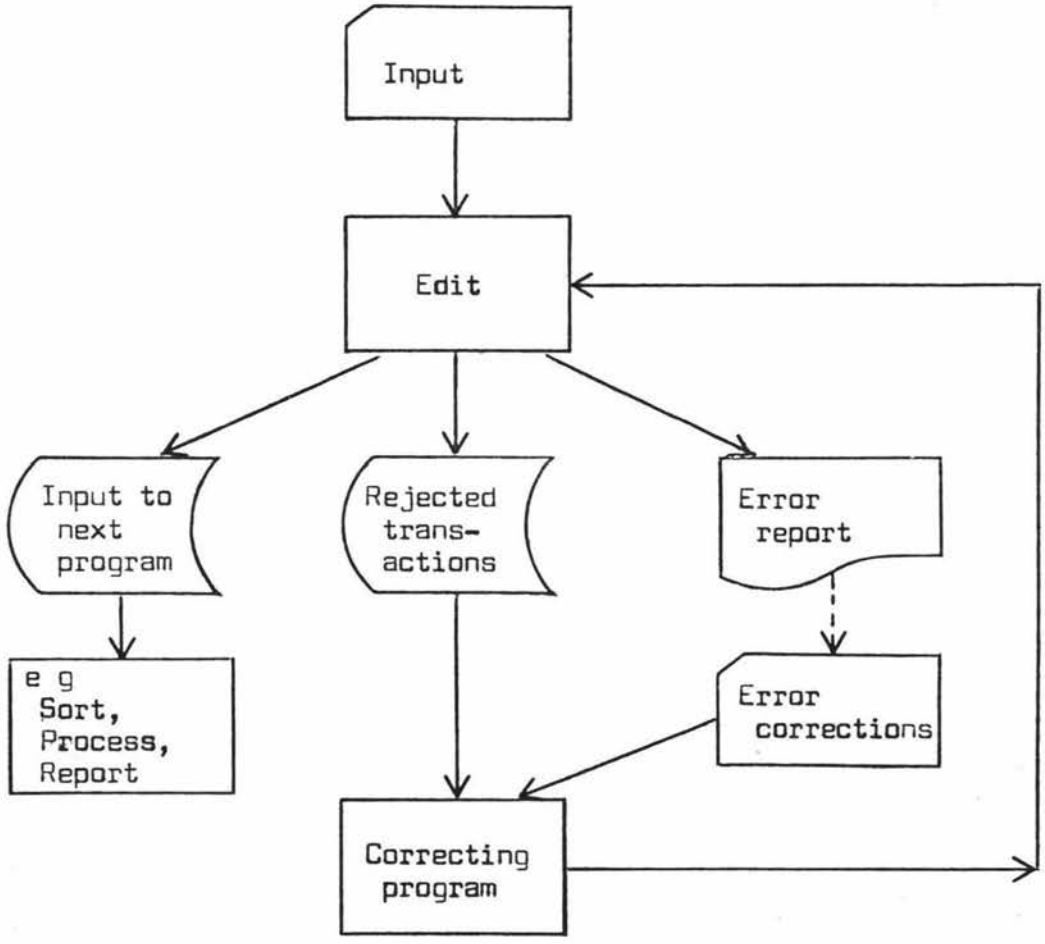


Fig 3.5 Rejected errors recycling.

Data should also be checked for validity at the stage it is used to update master files and at the stage it is converted to report information. Some hardware and software failures can also be trapped by doing this.

Other important control techniques, that should be mentioned for completeness, are listed here:

- Adequate documentation of software modifications should be kept.
- Copies of important files should be backed-up.
- New or modified software should be tested with data containing manufactured errors. (See Fig 3.6).

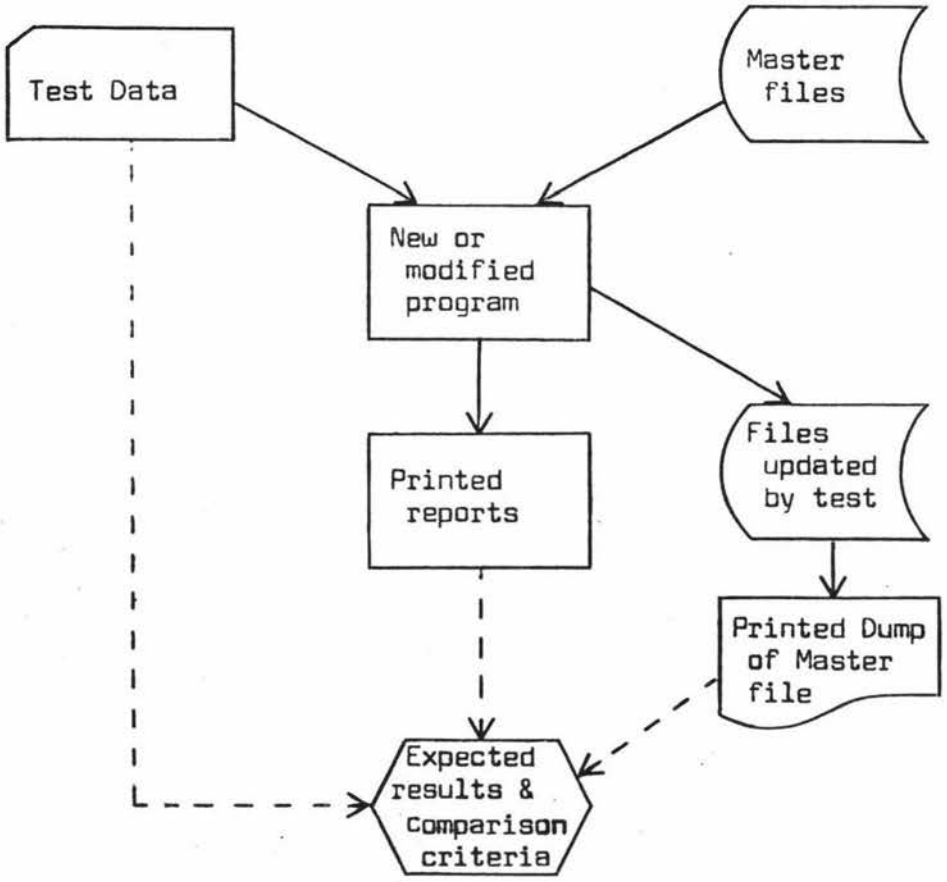


Fig 3.6 Use of test data for software evaluation.

After data has been applied to the master file, any further programmed error detection or correction is normally impracticable.

3.6 Summary

A major problem with computer data processing is control over the quality of the processing. This chapter has surveyed the major techniques and considerations associated with the different control stages. Adequate controls have a cost associated with them. The control over input is very important because most errors occur at this stage in the processing. Hardware is fairly reliable and, in general, has adequate controls. Using edit software is an effective control technique but is seldom sufficient on its own. Different methods fit different errors.

A detailed discussion on generalized editing follows.

4. ALTERNATIVE SOLUTIONS TO EDITING PROBLEMS

Discussed in chapter two were some of the advantages of using generalized software to solve generalized problems. Editing is a generalized problem. Its nature was described in chapter three. In this chapter, some of the alternative solutions to the problem of editing are outlined.

4.1 Interactive Entry Controlled by Miniprocessors

The populatity of using Audit Entry type equipment is increasing. This is because complex error checking can be performed at one of its sources without the assistance of mainframe processing. Detecting errors at their source is an advantage in itself. Avoiding the use of mainframe processing for all error checking is an important consideration for those installations whose data processing requirements approach computer usage capacity. Unfortunately, interactive entry equipment is unable to perform all the editing functions generally required. In particular the functions that cannot easily be performed are: interfield co-presence and co-limits, table membership of items, master file record correspondence, or any non-standard user required function. Some mainframe or programmable miniprocessor editing is still necessary.

Many data processing sites do not have this type of equipment. The cost of changing from their current devices is often prohibitively high.

4.2 User Written Editors

This alternative solution is the most commonly practised. Customarily, whenever a new data processing application is

implemented for a business, an editor must be written. Although editors are usually straightforward programs to write, all the problems and inadequacies of custom-built software, discussed in chapter two, can be expected. Specifically there will be the costs of development; time and labour, and the costs of maintenance and enhancement support.

4.3 Generalized Editors

There is a very definite need for this type of software. To date much attention has been given to the development of other generalized software such as: sorting routines, data management systems, and reporters. A possible reason for this is that these problem areas mentioned are more interesting and more urgent. They directly assist the user to achieve his processing objectives. Perhaps it has taken a long time to fully realize the commonality existing between editors. The fact remains that generalized software to perform the editing operation is not widely available.

Editing has all the characteristics suitable for generalization. It is a common problem shared by many users. Most editors perform similar types of checks. Considerable programmer effort is inefficiently used re-writing editors containing only minor specific differences. These variations along with the objectives of providing portability, modularity, and flexibility, are well within the scope of a generalized editing solution.

5. OVERALL REQUIREMENTS OF GENERALIZED EDITING

5.1 General Edit Tasks

The purpose of this section is to isolate and explain the tasks common to the majority of specific editors. All these tasks need to be included within the functions of a generalized editor. In brief, the topics that will be expanded are:- the file environment of the editor,

- the types of error checks performed at different levels, and
- the report functions of the editor.

5.1.1 The file environment of the editor

In its basic form, an editor will utilize three files, (as in Fig 5.1). Firstly there will be a primary input file of data to be validated, often a card file. Secondly there will be an output file of clean data records directed to some auxiliary storage media, usually disc. This file is intended to be used as input to further processing. Thirdly a report file will be produced on the site line printer. This will be a listing of the input records with appropriate messages accompanying any erroneous data detected. Mandatory to an editor will be the primary input file and one of the two output files. The data read could originate from any type of device capable of acting as an input peripheral. Similarly output files could be directed to any device capable of acting as an output peripheral. (See Fig 5.2).

A more comprehensive editor will need to allow master file record correspondence checking and error recycling. The term 'error recycling' refers to the process of saving rejected transactions in an auxiliary storage file, pending correction and re-editing. In figure 5.3, the on-line master file is shown as FILE 4, and the error recycling files

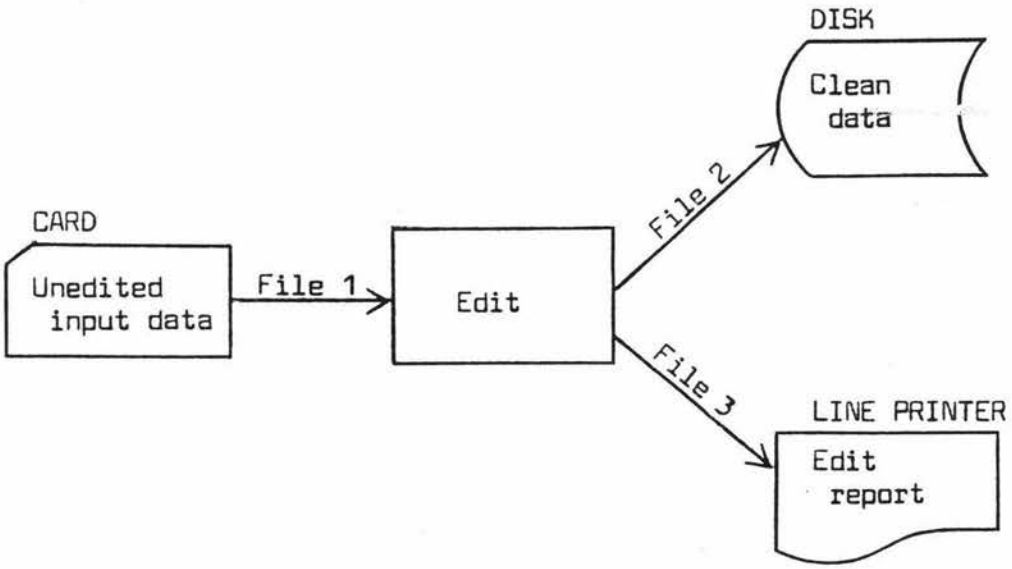


Fig 5.1 Three files utilized by a basic batch-processing editor

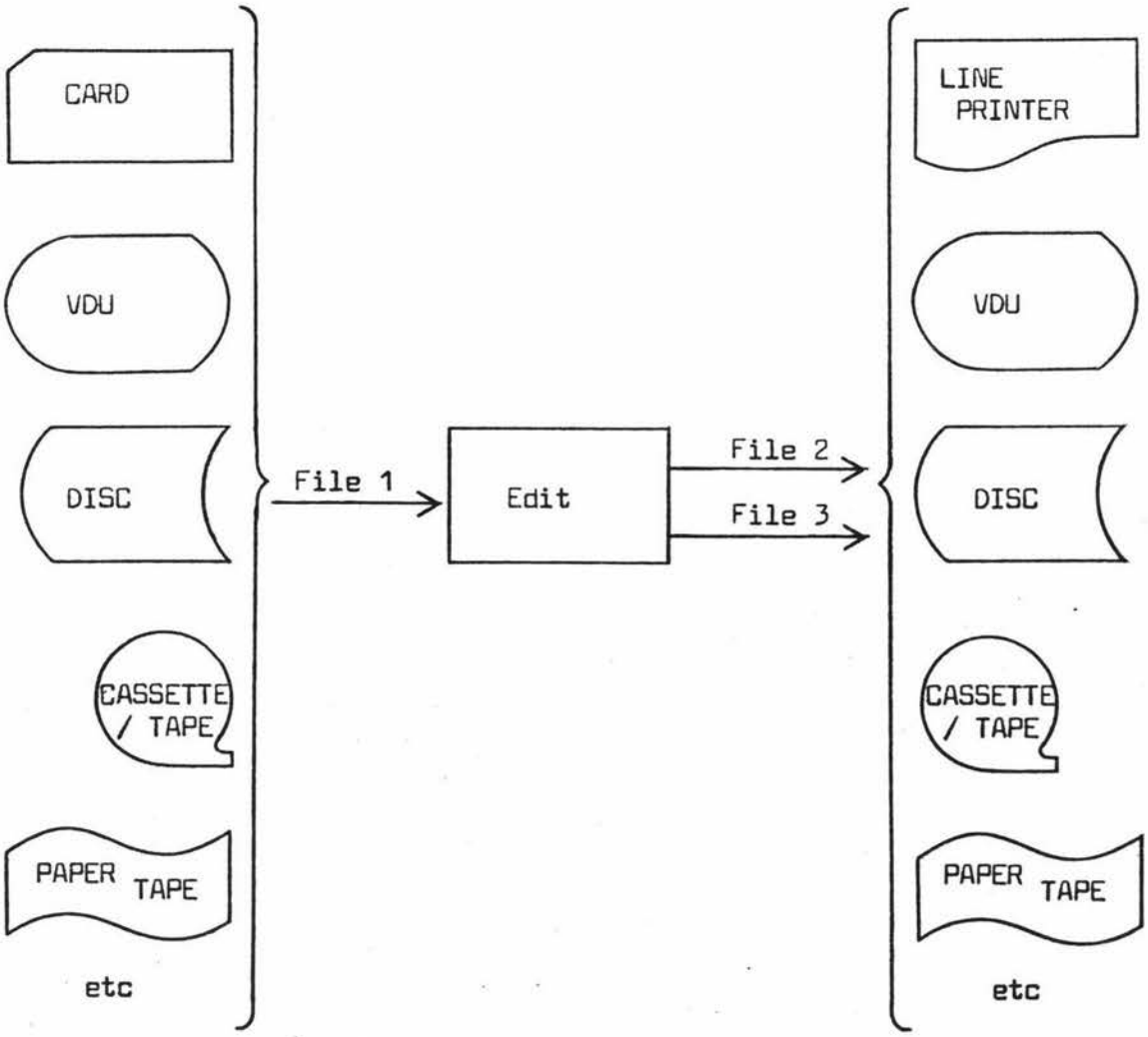


Fig 5.2 Some of the appropriate Input/Output devices that may be used.

are shown as FILE 5-O and FILE 5-I. Correcting the errors-file, (rejected transactions) may be an edit function or may be performed by a separate program.

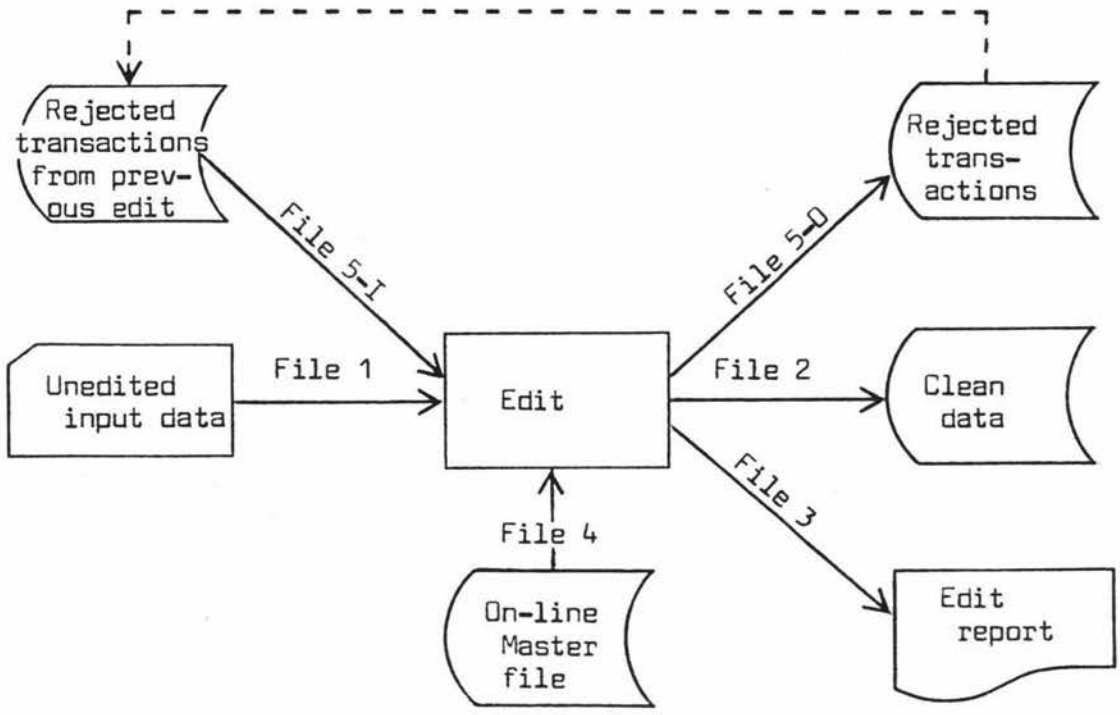


Fig 5.3 File environment of the edit with on-line master file and error recycling files.

Further enhancements to the basic editor will provide multiple report generation. For example, separate listings of the clean data, the rejected transactions, and the edit run statistics, will be possible. In this case the editor would need to use subsequent disk to printer utilities or rely on operating system output spooling.

Described so far are the general forms of an editor run as a master file pre-update and batch processing job. The term 'master file

pre-update' is in this context taken to mean editing will be completed before any further processing is performed, (e g actual updating of master files). It is also typical for editing to be performed in a master file permanent-update mode where records in the master file are 'updated-in-place'. Here the editor will be used as a subroutine procedure. It validates a transaction between the reading of that transaction record, and its use to update the appropriate master file record. A clean data file will not be produced in the permanent-update mode. (See Fig 5.4).

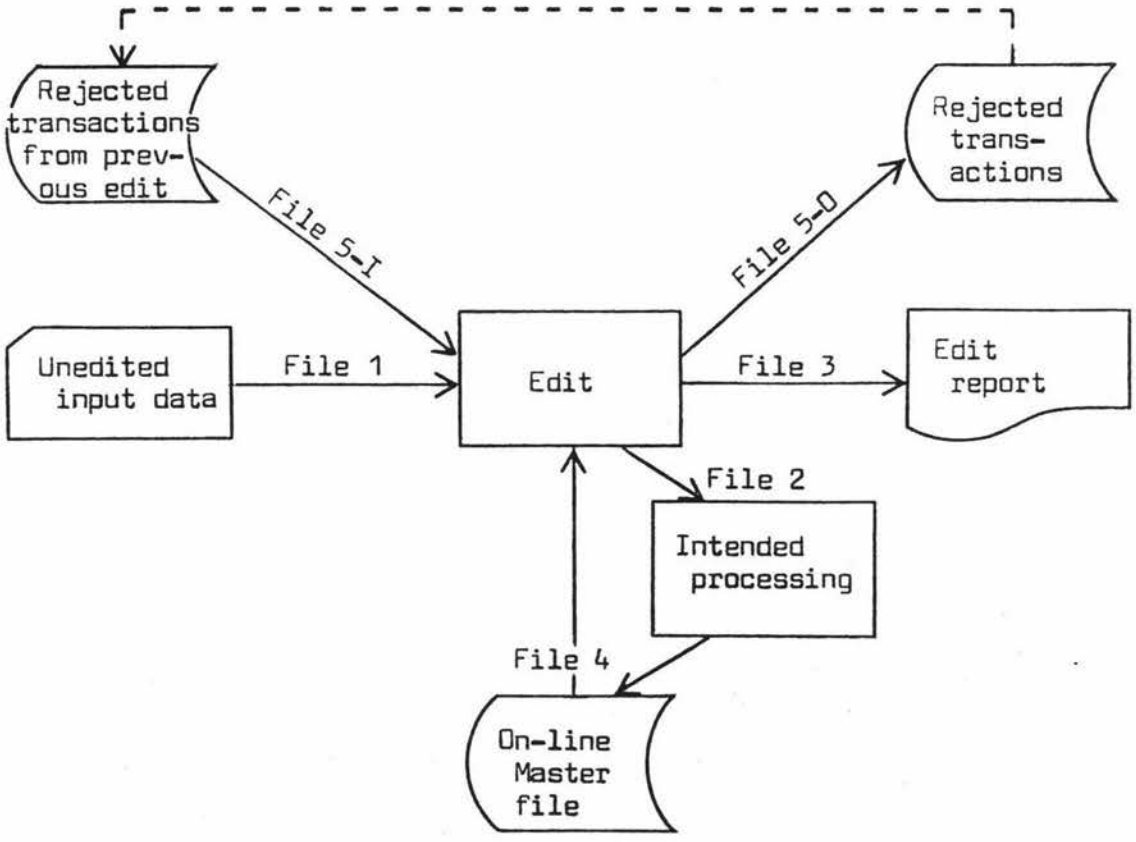


Fig 5.4 File environment for permanent-update edit.

Both pre-update and permanent-update modes of editing can be used under batch or on-line processing. Usually though, pre-update editing is associated with batch processing and permanent-update editing with

on-line processing. Figure 5.5 shows the file environment of a general editor under on-line processing. Note that basic on-line editing does not include the error recycling files or the edit report files. These can be included as well if necessary.

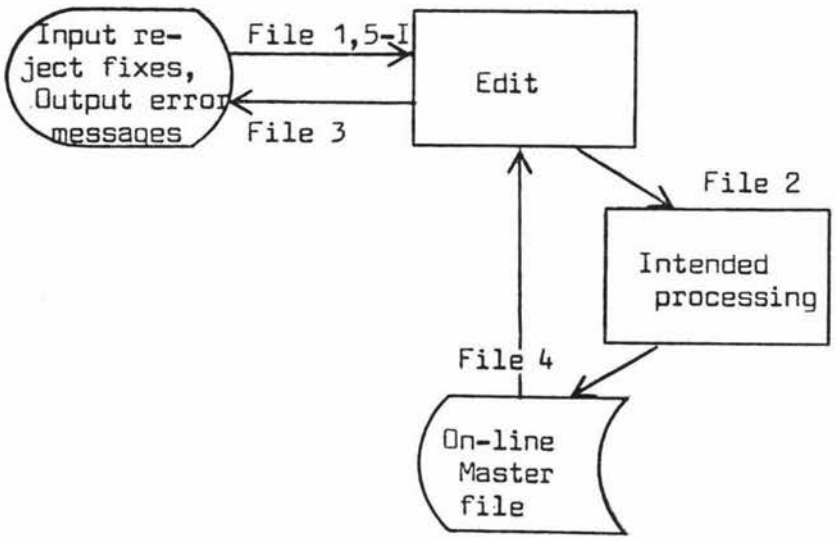


Fig 5.5 File environment for basic on-line edit.

In brief, a general editor will have some or all five of the files listed below:

FILE 1 - primary input of unedited records,

FILE 2 - output of clean transaction records,

FILE 3 - output of Edit reports,

FILE 4 - online master file, and

FILE 5-O - output of rejected transactions

5-I - secondary input of corrected transactions from previous edit.

5.1.2 The types of error checks performed at different levels

There is a very definite hierarchy of error-checks that will be functions of a general editor. The levels were identified in section 3.5.4, (Error control by software), to be: file, batch, transaction, and field. Set out in this section are descriptions and examples of the checks that will be made at each level. Those ones mentioned may not be inclusive of all editing checks possible, but they will be typical of the ones expected in the first version of a general editor. They are presented in lowest to highest level order.

(1) Field level checks

A field is the subdivision of a record which can be used to represent an item of information. In this definition, 'field' is synonymous with the term 'data item'. Records are comprised of one or more related fields. In COBOL terminology - 'field' is taken to include both 'group' and 'elementary' fields (or data items).

Checks performed at this level can be very effective in locating transposition and substitution errors.

a. Class:

(description)

There are three standard classes of data, numeric, alphabetic, and alphanumeric. Each class is a subset of a computer's character set.

(i)	Numeric	-	the digits 0 to 9, and optional leading or trailing + or -
(ii)	Alphabetic	-	the letters A to Z
(iii)	Alphanumeric	-	0 to 9, A to Z, and ∅ (space).

(example)

	416	A numeric class check performed on this list
	664	would locate the shift error (K for 5) and the
ERROR →	6K3	transposition error (Z for 2).
	522	
ERROR →	Z48	

(problems and exceptions)

This check is more complex if the item within a field is left or right justified and the trailing or leading spaces must be skipped.

The user may wish to define new classes of characters or ordered combinations of characters.

e g HD8639 - (Car registration number) Two letters followed by up to four numeric digits.

CARD-CODE - (COBOL data-name) Alphanumerics with possible hyphens inserted but not at either end.

$7,483,115 - Left most character is '$' and ',' inserted between triple digits.

A numeric item cannot consist of a sign character only.

Signed numeric items may have spaces between the sign and the most significant digit, (e g -__6).

b. Justification:

(description)

Items within a field may have a justification requirement. Normally numeric items are right justified and alphanumeric items are left justified.

(example)

	416	KAKAHI QUEEN	Justification checking can
	664	SILVER HOFLANDS	help detect errors like:
ERROR →	653_	PURPLE GROUSE	records displaced or items
	522	BLACK HACKLE	not starting in the correct
ERROR →	248	_DORITHY - RED	field.

(problems and exceptions)

Items incorrectly justified, like 'PURPLE' shown above, cannot be detected using this check.

c. Presence:

(description)

There may be a requirement that at least some characters exist within a field.

(example)

	416	KAKAHI QUEEN	Presence checking can help
	664	SILVER HOFLANDS	detect missing-item errors.
ERROR →	___	PURPLE GROUSE	
	522	BLACK HACKLE	
ERROR →	248	___	

(problems and exceptions)

Some fields may have a requirement to be left empty, like the one above containing the first letter of 'PURPLE'.

If a numeric field contains all zeros, it may be considered not present. (A range check excluding zero could solve this problem).

d. Range:

(description)

Number ranges are often referred to as 'limits of reasonableness'. As this phrase suggests, items found to lie outside an expected range can be regarded as unreasonable and therefore erroneous.

(example)

	416	
ERROR →	964	If the item-no. was defined to have values
	653	between 100 and 800, then the two errors shown
	522	would be detected best by range checking.
ERROR →	048	

(problems and exceptions)

Items may be alphanumeric and still require range checking. Most computer systems cater for string comparisons.

Alphanumeric or numeric range checking needs to take into account the possibility that the items may not be justified.

Fields containing dates may require range checking. Sometimes the upper limit may be the current date, (i e the run-date of the edit).

e. Date:

(description)

A common item included as a record field is a date. Checking for impossible dates is a valid editing function.

(example)

	416	100874	
ERROR →	664	061472	Impossibe month-numbers, and
ERROR →	653	310972	day-numbers can be detected using
	522	010133	a date check.
	248	190276	

(problems and exceptions)

There are several different conventions used for representing dates. They include putting the month number before the day number and using the character '.', '/', or '-' to separate day from month and month from year. Century number is sometimes used. Month names and month name abbreviations are also commonly used.

Here are several other ways 100874 can be represented:-

- 081074 (American notation)
- 10/8/74 (One digit only for month)
- 10TH AUG 1974
- 10.8.74

f. Table membership:

(description)

This form of check is a more general case of the range check. Rather than having one pair of bounds, there is a whole series of bound pairs for the item value. These bounds are typically represented by making every possible item value a member stored in some memory table.

Table membership checking can be used for alphanumeric items.

(example)

Membership table values:
100, 220, 246, 247, 248, 249, 461, 658, 659, 660, 661, 662, 663, 664

Values checked:

ERROR →	416	
	664	A transposition and a transcription
ERROR →	653	error or a completely erroneous value
ERROR →	522	can be detected using a table membership
	248	check.

(problems and exceptions)

Keeping the table updated may be difficult and expensive for those cases where the information it contains is volatile (e g the registration numbers of a vehicle population, or the names of ships in port).

Alphanumeric table membership checking can be unsuccessful in some cases. To a computer, 'McNamara' and 'MacNamara', 'New York' and 'NewYork', or 'Mathematics' and 'mathematics' are pairs of different symbols. Rejecting a transaction and possibly a whole batch of transactions because a minor spelling difference is detected can be both irritating and expensive.

g. Check digit:

(description)

In data processing systems numbers are used for practically all identification, (e g tax, account, and customer numbers). The identification number is usually the most important item of a transaction record and its correctness is vital. One of the most efficient and effective methods available for checking field items is to attach to it an additional digit called the check digit. It is compiled as some function of the remaining digits in the identification number.

Alphanumeric fields can include a check digit also but this is not a common practice.

Various formulae may be used for calculating check digits. Two formulas are called Modulus 10 and Modulus 11. The latter is more common.

As an example, the number 423-055-c, where c is the check digit, will be used to demonstrate each formula.

The calculation for Modulus 10 is done as follows:

(i) Eliminate all non numeric elements (i e -, / etc).

(ii) Double every alternate digit starting with the units position, and multiply the others by one

4	2	3	0	5	5
x1	x2	x1	x2	x1	x2
4	4	3	0	5	10

(iii) Add all the digits of the products together:

4 + 4 + 3 + 0 + 5 + 1 + 0 = 17

(iv) Subtract the result from the next highest multiple of ten to arrive at the check digit:-

$20 - 17 = 3 \rightarrow c$

(v) Reposition non-numeric elements, and append the check digit.

The complete identification number is then: 423-055-3.

The Modulus 11 calculation proceeds as follows:

(i) Eliminate all non-numeric elements.

(ii) Multiply each digit of the basic number by a factor that corresponds to the position of that digit:

4	2	3	0	5	5
x1	x2	x3	x4	x5	x6
4	4	9	0	25	30

(iii) Add the products together:

$4 + 4 + 9 + 0 + 25 + 30 = 72$

(iv) Divide the result by 11; the remainder is the check digit.

$72/11 = 6$ with remainder of 6

$6 \rightarrow c$

(v) Reposition non-numeric elements, and append the check digit.

The complete identification number is then: 423-055-6.

As stated in section 3.5.2, approdimately 97% of transposition and substitution errors can be detected using check-digit validation [5].

(example).

(Using Modulus 11 check digit method)

ERROR →	461-4	c f	calculated c = 8;	4 is valid for 416
	664-8	c f	calculated c = 8;	OK
	653-3	c f	calculated c = 3;	OK
	522-4	c f	calculated c = 4;	OK
ERROR →	243-1	c f	calculated c = 8;	1 is valid for 248

(problems and exceptions)

Referring to the example above, it will be seen that 461, 664, 243 all generate the same check digit showing that this checking method is not foolproof. It is unlikely that 461 say, will be mistakenly recorded as 664, although this possibility can not be discounted. Perhaps a more likely example is the case where a double transposition occurs and the check digit formula being used is Modulus 10; e g 7846 and 4678 will both share the same check digit as will 7846 and 4876.

Check digit calculation has a cost associated with it; an extra key stroke or transcribing operation is required and an extra position of the transaction record is taken up.

From the generalized editing point of view, there is a problem deciding which formula to offer for check digit checking. Unfortunately this is compounded by the realization that several different versions of each formula are commonly used. Eighteen different schemes are described in one manual [4].

In one reference [3], rule one of the Modulus 10 method is stated:

"1. The unit's position and every alternative position of the basic code number are multiplied by two.",

while in another reference [6], it is stated:

"1. Double every other digit of the basic number, and multiply the other digits by one:

3	7	6	9	2	5
x2	x1	x2	x1	x2	x1
6	7	12	9	4	5

This example shows that the digit next to the unit's position is the first to be multiplied by two. A different check-digit will result.

The Modulus 11 methods also vary. Using the same two references [3] and [6], rules 1 and 3 are shown below for comparison.

[3]

"1. Assign a weight (checking factor) to each digit position of the basic number. The weights to be used are ... 2, 3, 4, 5, 6, 7, 2, 3, 4 starting with the unit's position of the number and progressing toward the high order digit."

"3. Since this is a Modulus 11 System, divide the sum of the products by eleven, and subtract the remainder from eleven."*

c f [6]

"1. Multiply each digit of the basic number by a factor that corresponds to the position of that digit in the account number."

"3. Divide the result by 11; the remainder is the check digit. If there is no remainder, the check digit is 0."

* Note.
This method fails if the remainder is 0 or 1. Subtracting this then from 11 would give a two digit result of 11 or 10, although this may be represented as, for example, A and B (i e Hexidecimal notation).

(2) Transaction level checks

A transaction, often called an input record, is a collection of related items of data. A destinction needs to be drawn between physical and logical records. The unit of data for input or output is called a physical record. One or more of these make up a logical record. The terms 'transaction' and 'record' are commonly used to name a logical record, and this use will be assumed throughout this thesis.

After field checking the higher level transaction checks can be made. These are checks performed on inter field relationships; field copresence, field sequencing, and transaction construction.

Note. Transaction level checks involving master file record matching or retreival are dealt with for convenience under the 'file level' heading.

a. Transaction construction:

(description)

If a transaction is comprised of several physical records, checks need to be made to insure completeness and orderliness. Physical records missing or out of order can then be detected.

(example)

(Each transaction is comprised of an (I)d record, an (A)ddress record, and up to four (S)ubject records.)

```
          I 7875118 V KELLY
          A MARYBANK RD 2;MANUKAU CITY;
          S 60.202
          S 60.203
          S 58.101
ERROR →   I 7365683 P A RAE                    (Address card missing)
          S 58.203
          I 7763140 C J TAYLOR
          A 44 AGRA CRESCENT; FEATHERSTON;
ERROR →   A 9 VIMY PLACE; LOWER HUTT;          (Second address card)
          S 36.208
          S 60.111
          S 34.101
          S 41.101
ERROR →   S 58.622                             (More than four sub-
                                                ject records)
```

(problems and exceptions)

Note that the number of subsidiary records expected, (in this case 'S' records; with a maximun of 4) may be given as a sub-field of, for example, the Id Record. In this case, the 'Transaction Construction' test can only take place, if a valid 'Number of S Records expected' field exists, and a relationship between field level, and transaction level checks is required.

b. Field sequencing:

(description)

When the order of records is important, sequence checking is a normal editing function. The sequence of records can have different forms. Sequence fields can be increasing or decreasing. Their sequence can be in discreet values or just in some absolute order. Transactions that are missing, duplicated, or out-of-order may need to be rejected and can be detected with the appropriate sequence check.

(example)

	1064	
ERROR →	1065	(1066 missing)
	1067	
	1069	
ERROR →	1069	(1069 duplicated)
ERROR →	1068	(1068 out-of-order)

(problems and exceptions)

With absolute ordering, duplicate sequences may be permissible.

Note, that in this example, 1068 would be flagged as both missing and out-of-order.

c. Copresence:

(description)

There are two basic forms of copresence checking; copresence existence and copresence limits. The first form is applicable when the presence or absence of a field depends on the presence or absence of another. The second form is more complicated. Here the range of values in a field depend on the value, absence or presence of another. Correctness of the information itself is tested by this technique. Inconsistencies found between related items can often be traced back to an incorrect recording operation.

(example)

(Copresence existence)

		1	2	3	
	7357834	58101	60111		
ERROR →	7662378	58201		77102	(If paper 3 exists, then paper 2 must also exist)
	7860536	58101	14103		
ERROR →	7469804		60206	58202	(If paper 2 exists, then paper 1 must also exist)
	7754574	77101	58202		

(Copresence limits)

	Batch No.	No. Eggs	No. Hatched	No. Dead	
	237	156	152	4	
ERROR →	238	159	160	0	(More Hatched than No. Eggs)
ERROR →	239	143	140	8	(No. Eggs ≠ No. Hatched + No. Dead)
ERROR →	240	160	159	0	(No. Eggs ≠ No. Hatched + No. Dead)

(problems and exceptions)

Copresence conditions can become fairly complex involving, the logical operators AND, OR, and NOT, the relational operators =, < , > , the arithmetic operators +, -, *, /, or table membership of items.

e g IF date of Birth > 1930

OR (salary ≥ 5,000 AND salary ≤ 2,5000)

AND position ≠ DIRECTOR

AND NOT name member of VIPLIST

THEN salary increase ≤ 10/100 * present salary

It may be difficult for a generalized editor to support anything but fairly simple copresence conditions.

(3) Batch level checks

A batch is a group of records having a common identity. Usually each batch contains a manageable number of records; a specific number or a convenient number collected for a given unit of time or work. Batching helps to confine the scope of errors and to assist in their location. Examples include: missing transactions, duplicate transactions, or incorrectly transcribed values. Applying batch total checks is generally used for batch editing (i e not for on-line editing operations), and it is a standard and widely used data processing procedure.

a. Batch totals:

(description)

Three general types of totals can be used with batched transactions; control totals, hash totals, and document counts.

- A Control total is a simple sum of the same item from each transaction. The total can be a financial sum for the batch or a quantitative amount for the batch i e it is a meaningful value.
- A hash total is also a simple sum of the same item from each transaction. In contrast to a control total, it does not represent a meaningful value. The summation of identification or key fields in a batch is a good example of a hash total.
- A document count, as the term suggests, is a simple count of the transactions or records in a batch. A correct document count may be a specific value or it may have a range of values.

Typically each batch is accompanied by a special form (transmittal-control route slip shown in the example below), containing its

precalculated control totals, and the first and last transaction identification values. This information is usually input before the batch of transactions. When the end of the batch is reached, the entered pre-calculated control totals and identification values are compared with those computed as each transaction was being edited. Any discrepancies will be noted on the edit report. The entire batch will normally be prohibited from further processing.

(example)

Transmittal-control route slip

Batch No.	144	To	Receiving Dept.
Date	17/6/78	From	Purchasing Dept.
No. of documents	5	Numbered From 32155	To 32603
Control totals	669.20, 8		
Hash totals	160797		

Transactions

Acc No.	Balance	No. deposits
32155	10.55	2
32156	106.24	1
32160	27.68	0
32162	499.10	4
32164	25.63	0
160797	669.20	8

Batch comparisons

	Calculated	Computed	
First ID	32155	32155	
Last ID	32603	32604	← ERROR
Document count	5	5	
Control tot. (Bal.)	669.20	669.20	
Control tot. (Dep.)	8	7	← ERROR
Hash tot. (Acc No.)	160797	160797	

(problems and exceptions)

If batch total checking is performed, then as each transaction in the batch is edited, it must be saved in a temporary file. When the batch totals have been validated, the batch of transactions in the temporary file need to be transferred either to the clean data file or to the errors file.

Batch validating is complicated by the possibility of field level or transaction level errors particularly if they involve the field items required for totalling. Normally the entire batch is rejected if any errors are detected at any of the checking levels, but users may occasionally wish to let valid records through.

In attempting to locate the cause of a batch error it is important to realize that the precalculated totals may themselves be in error.

As with the check digit editing method, two batch errors may result in the totals comparing correctly, although this has a low probability.

On-line (i e real-time) batch checking can be used, but in a slightly different way. The batch checking information and the batch of transactions are first keyed in.

When the end of the batch is reached the pre-entered values are still compared with those that are computed, so that discrepancies can be reported. At that stage, however, the batch may have lost its identity. All the transactions will have been applied to the master file and it will be too late to reject them if any batch errors were detected. Any batch error messages produced by an on-line edit can be used to help correct the master file perhaps with an update program, run sometime later.

An alternative, and possibly more typical approach to on-line batching, is to save the transactions in a temporary file as they are being entered and edited. Before the next batch of transactions are accepted the batch checks are performed. If the batch is found to be complete, then it is applied to the master file. If the batch is found to be incorrect a choice is made by the terminal operator whether the batch should be reviewed and corrected immediately or appended to the rejections-pending file.

(4) File level checks

A file is a collection of related records. Each record is usually related by a common type of identity and several common types of item information. Normally the structure of relationships is kept by using an identical position within each record for a given item.

When editing, file presence and master file record correspondence can be performed as file level checks.

a. File presence:

The most basic general editor requires the presence of the primary file of unedited input data. Editors for more complex file environments, require the presence or absence of various other files. Sometimes these presence conditions are quite complex, for example, the clean-data file may be present from previous edits and should be made ready for the addition of more clean transactions.

The rejected-transactions file also may be present. Its version number may need to be used so that a new file can be created. It might also be read as a secondary input, corrected and then re-edited. The presence or absence of any file may depend on its serial-number, version-number, or date of creation/last access.

Note that this form of check helps to detect operation mistakes and not errors within the input data. This does not mean it is not a valid type of check. Providing the correct file environment is important not only for correct running of the edit but also for safeguarding against the accidental corruption or destruction (by overwriting) of existing files.

There are methods of ensuring file presence and thereby providing the correct file environment to the editor. One method is to rely on the computer operations staff to load the correct tapes, cartridges, diskpacks, etc, or copy the correct files on to the on-line auxiliary storage devices, before the editor is run. This is often done where job submission forms accompany programs. (See Fig 5.6). The job submission form information includes a list of files required by the program. A second method commonly used on multiprogramming systems is to use the operating system file management facilities. These are evoked through the job control language and are used to correctly sequence the availability of files to a program as they are required. In this case, if necessary, the operator will be instructed, through the console, which files should be made available to the system. On receipt of these files the operating system will insure they have the correct title and serial-numbers. Finally, a third method of file management is to program it into the software itself. This is fine but not an appropriate method for generalized software. The organization and management of files is extremely hardware dependent and the high-level language constructs provided are often installation specific.

In the interests of producing portable software written in a high level language, it is not feasible to generalize file management so that it can be provided as a generalized editing facility.

JOB RUN SHEET

1. Job Details:

Job Name/Number	Contract No.	Date	Sheet No.
M00-419-1710	168212	11/8/72	1

2. Scheduling Details:

Computer	Processing Mode	Run Time	Max. Core	No. of Tape Decks	Dispatch Box Number
ICL 1904A	TESTING	00.50	15 K	4	419 JONES

3. Magnetic Tape Details:

TAPE INPUT		TAPE OUTPUT		Comments
Unit/File Name	Reel No.	Unit/File Name	Reel No.	
FB9A-INPUT	3211	FB9A-MAST (1/0)	P00L	
FB9A-MAST (0/0)	3072	(1/1)	P00L	
(0/1)	4601	(1/2)	P00L	
		FSSE	MHWK	

4. Operating Notes:

5. Run Control:

SCHEDULING AUTHORITY		PRIORITY AUTHORISATION AND RESULTS OF RUN	
Run No.	Approved by	Operator's Date of Run	Log Details/Operator Comment
1			
2			
3			
4			
5			

6. Attach Labels for Print Tapes.

Fig 5.6 Sample job submission form.

b. Master file record correspondence:

The control identifier (the key field) of the transaction record being edited is used to test the presence or absence of the corresponding record in the master file. Master file record correspondence checking has been included in this section because it is an inter-file comparison. The check itself is performed at the transaction level. It is similar in nature to the table membership test if the table was a file. Membership (i e presence) is a valid edit criterion they both share.

The usefulness of this type of check depends on the type of transactions being edited. If the information is to be used to update fields of the master file records then absence of the master file record indicates either a master file error or an invalid key given in the transaction. If the transaction is to be used to insert a new master file record then presence of the record indicates a file or transaction error and, like the file presence check, it can help prevent loss of information by over-writing.

Most generalized and specialized software used for information, retrieval or master file updating and maintenance, provides record correspondence checking. If this software is used in conjunction with an editor in an on-line or permanent update mode, (as described in section 5.1.1), then using the edit correspondence check as well may not be necessary. It is used to best advantage in the pre-update mode, because the edit check is then a type of simulated update to reveal problems with the data that would otherwise not appear until much later in the processing. Not only are the errors detected earlier, but also they are detected with all the other forms of data errors. Reducing the number of occasions where errors need to

be searched for and corrected is an efficient systems procedure. It is an appropriate and useful edit function.

Mention should be made of a situation where the master file record correspondence check can fail. If during an edit run a transaction is present for up-dating a master file record and it precedes the transaction to create that master file record in the same run, then the first transaction will be rejected as having no master file record correspondence. This problem is also unavoidable. Presorting the transactions is not a good solution because the sort keys have not been validated and the order of input transactions will be lost. (Maintaining input order may be a control requirement).

Note. Once the transaction and master file records have been matched, all the other transaction level checks (copresence etc.,) can be extended to include master file fields as well as transaction fields.

5.1.3 The report functions of the editor

As discussed in section 5.1.1, (The file environment of the editor) reporting is a major function of editing. Normally the editor will produce a report of all input transactions or at least identify those that have been rejected. A report listing the clean data may also be a requirement. On-line editing necessitates direct communication from the editor to the operator at the terminal (VDU).

Reports generated by an editor are typically used as an audit trail, (i e transaction log) and as a reference while correcting rejected transactions. In some situations they can even be used as managerial reports. Under on-line processing they can greatly assist immediate error correction during the data capture operation.

Following are more detailed lists of the types of information that may appear on the different edit reports as run under different modes of operation.

(1) Pre update edit, run as a batch processing job.

a. Transaction report (Rejected transactions only, or all transactions).

(i) Report heading:

- Installation identification. (Often preprinted on the line-flow).
- System identification. (Program system the data is being edited for).
- Program identification. (That it is an editor).
- History. (Run date, serial number, etc).
- Other detail. (Department from/to, Operator, etc).
- Program options and defaults. (Type of report, etc).

(ii) Page heading:

- Title.
- Date.
- Page number.
- Batch number.
- First key on page. (Identification field from the first transaction to be listed).
- Column identification headings.

(iii) Batch heading:

- Batch number.
- Number of transactions in the batch.
- First and last Key for batch.

(iv) Transaction detail:

- Key. (Identification field).
- Selected items from the transaction (possibly formatted).
- (Possibly) error messages. (Text explaining the nature of the errors, item identification by a name or a position indicator).

(v) Batch footing:

- Batch number.
- Pre-entered batch information. (Number of transactions, first and last transaction keys, control totals, hash totals).
- Computed batch information.
- Any batch error messages.
- Number of transactions in error.
- Rejection notification. (Number of rejected transactions, number of accepted transactions, or 'total batch rejection').

(vi) Page footing:

- Batch number.
- Last key on page. (Identification field from the last transaction to be listed).

(vii) Report footing:

- Any grand total comparisons. (Totals of Batch totals etc).
- Total number of rejections. (Number of transactions, number of batches).
- Total number of acceptances. (Number of transactions, number of batches).
- Program run time statistics. (Execution time, frequency of checks).
- Program completion notification.

b. Clean data report

(As above but without any rejection detail).

Note. The clean data report may contain unprocessed transactions output from previous edit runs.

(2) Permanent update, run as an on-line processing job.

a. Run-time report

(This report is produced interactively and will be interspersed with detail keyed by the operator. The detail generated will be meaningful in this context).

(i) Report heading:

- (Possibly) installation, system, and program identification.
- Operator prompt for setting program options or displaying options menu.
- (Possibly) prompt for history information. (date, serial, operator identification, department number etc).

(ii) Screen headings:

- (either) record-type and column headings, (or) item-name prompts.

(iii) Screen detail:

- (Possibly) error messages.

(iv) Report footing:

- Any grand total comparisons.
- Total number of rejections pending correction. (Some errors detected at the data capture stage can not be corrected immediately).
- Total number of acceptances.
- Program completion notification.

b. Hard-copy transaction report

(This report may be simply a hard-copy of all communication to or from the terminal including re-entered transactions or may be a complete report as described for an edit run under batch processing. If a full transaction report is produced, some of the information may not be included. For example, batch headings, batch footings, and rejection statistics may not be relevant).

5.2 Facilities Provided for User Exceptions

If all the general edit tasks described in section 5.1, are provided as facilities of a generalized editor, no guarantee can be given that all the needs of all users are catered for. In fact, it is quite unlikely that there are provisions available for all the needs of one user. Those needs that are not provided for, are named 'user exceptions' by this thesis. If software was generalized sufficiently to incorporate all user exception requirements, then this would be at a high cost, even if it were at all practicable. The software would be slower to run, require more machine space to work in, and it could be more complicated to use. These additional costs are likely to deter the user from using this software. On the other hand, if the user cannot fit the software to his exceptions, he is just as likely to be detracted from its use. What is the solution?

5.2.1 Alternatives for providing user exceptions

The first alternative is to do nothing and allow the user to customize the generalized software to his needs. A source version of the program must be available to him for this purpose. Also, he must have the resources to make his changes, for example: skills, time, equipment, and the compiler needed to translate his modified source back into machine code.

If this alternative is used, then the software will lose some degree of its generality after it is modified. More importantly, any enhanced versions of the software he receives need to have his former modifications reapplied.

A second alternative to solving this problem is to equip the generalized software with a mechanism to evoke user provided procedures (subprograms) at stages throughout the run, wherever user exceptions are required. This method is attractive because complete generality is maintained. The forms of generalized software discussed in chapter two, except program generators, would need to use some form of external-code-linkaging to connect the user procedures with the main routine. This technique is machine dependent, is not well defined in high level languages, (e g some COBOL compilers use the 'LINKAGE SECTION', others use the 'DECLARATIVES'), and as a result, will not help the software's degree of portability. With program generators, the problem of user procedure inclusion is easily solved. The user procedures are simply included to be compiled with the rest of the generated program.

Choosing this scheme is appropriate for allowing the many user exceptions that are possible under a generalized editor.

5.2.2 Examples of generalized editing user exceptions

It is possible that user exceptions will be needed for most of the edit tasks described in section 5.1, i e users may require the editor to work under a different file environment, provide different or more complex error checks, and perform quite different reporting functions. A few examples are listed below under each of the general edit task types.

(1) File environment exceptions

- The selection of input records to be edited may be more complex than just reading the next one in sequence.
- Each record may need to be reformatted before it is edited.

(e g variable length fields changed to fixed length ones).

- The routines for correcting the recycled transactions could be evoked by the editor before they are validated.
- Appending new transactions to the clean data file may require installation specific constructs.
- The on-line master file may be a complex database requiring controlled access.

(2) Error check exceptions

- Some users may wish to check the relative position or number of occurrences of a particular character within a field.
- The version of check digit calculation may not be available as an option of the editor.
- Complex copresence checks may be important.
- Line control calculating could be necessary.

(3) Reporting exceptions

- Unusual hardware device characteristics may not have been provided for in the general reporting function.
- Additional formatting may be desired.

6. DESIGN CRITERIA FOR GENERALIZED EDITING

6.1 Overall Design Objectives

In the interests of producing a useful generalized editor certain design criteria must be met. Portability, modularity, flexibility, and ease of use are the design criteria discussed in this chapter. If these objectives are adequately achieved, then the user can expect the following advantages:

- There will be a saving in software development time, and software development costs.
- The reliability of the editor will be immediate.
- Programming personnel can be employed for other software development.
- The production of a better standard of software is likely, (i e, well structured with good documentation).
- The editor will be easier to use.
- The user exceptions will be easier to incorporate.

6.2 Portability

This term refers to the degree to which a program can be run without modification and successfully produce the same results on different types of computers. It also refers to the extent to which the language of the program is used throughout the industry. In other words, portability is not only a measure of the program's mobility but also a measure of the host language's popularity. The low-level assembler languages are generally not portable at all, but the high-level languages, such as BASIC, COBOL, FORTRAN, PL/1 etc, are portable to different extents.

It is hoped that the generalized editor described by this thesis could be shared with any number of users. As a result, portability is a major design objective for it.

6.2.1 The portability of different languages

Following is a brief survey of possible host languages to be used for implementing the generalized editor.

- ALGOL is a publication language. It has been standardized except for input/output constructs and has become popular mainly in Europe and for theoretical use.
- BASIC was developed as a student language. Although it has become widely accepted and is fairly mobile, its input/output facilities are very limited, it lacks many advanced features such as data structures, and it is not orientated to data processing problems.
- COBOL was developed as a language for business data processing. It has been standardized and has gained widespread acceptance, i e it is portable.
- FORTRAN evolved over a long period of time under some degree of standardization. (There are two standard versions of FORTRAN; BASIC FORTRAN and FORTRAN). Almost every computer with a memory size sufficient to implement FORTRAN has a version of it. It is not wholly suitable for business-type problems such as: file maintenance, data editing, and report production. Scientific and mathematical problems are its orientation.
- PL/1 is not yet very popular although it has a standard and can support data processing applications. Perhaps its complexities

detract from its use on computers other than the largest ones on which it can be implemented reasonably efficiently.

- RPG has no standard. It is intended for straightforward applications with uncomplicated logic and simple use of files. The language is especially suited for simple reporting problems. Because it can be implemented on all sizes of machines it has been widely accepted. IBM RPG II is the 'de facto' standard.

6.2.2 The portability of COBOL

The results of a language popularity survey, taken in the United States in 1977 [5], from 132 installations servicing over 4,000 users, show that COBOL is the top commercial programming language. Usage percentages are listed below in table 6.1.

It would appear that COBOL is the obvious choice as the host language for the generalized editor. Just how portable is COBOL?

Table 6.1 Programming language use in 1977

Language	Number of sites	Average %use where used	Overall %use
Assembler	97	22	16
APL	4	14	0.4
BASIC	11	15	1
COBOL	119	70	63
FORTRAN	45	9	3
PL/1	26	32	6
RPG	23	25	4
Other (ALGOL included)	42	16	5

Note. COBOL is used at 63% of the sites. It is used for 70% on the average, of all programming at those sites where it is installed.

COBOL was first conceived in 1959. Other versions were proposed in 1961, 1965, 1968 and 1974. The 1968 and 1974 versions were ratified as standards called ANS COBOL, (American National Standard COBOL). There are several differences between COBOL 68 and COBOL 74. For example, the COBOL 68 EXAMINE statement has been replaced in COBOL 74 by the more powerful INSPECT statement. The differences although slight are significant enough to mean that a COBOL 68 program may need modifying if it is to be compiled with COBOL 74.

In the interests of complete portability a subset union of COBOL 68 and COBOL 74 has been chosen for implementing the generalized editor. (The reserved word list and syntax of this subset have been included with this thesis as APPENDIX A and APPENDIX B respectively).

This decision is based on several factors. The COBOL 65 version compilers are unlikely to be used for much longer.

Certainly, the number of COBOL 74 versions being used at present is a minority, but this is expected to change. COBOL 74 being a larger language than COBOL 68 is not implemented on many of the smaller machines. It is important that generated editors be able to run on such smaller machines because they are the most commonly used in the industry. They are often referred to as medium systems.

The use of a subset of COBOL is not an absolute guarantee of portability. There are some constructs in COBOL that are not portable. For example, the names for hardware devices used for the assigning of files, depend on the COBOL implementation. To solve this problem COBOL has a CONFIGURATION SECTION in which all implementation dependent names can be collected together and assigned mnemonic names to be used in the body of the program. Other implementation

dependencies are also specified in this section.

6.2.3 The portability of the generalized editor

To gain maximum portability, the several implementation specifications listed below, should be referred to when developing the generalized editor.

- Write the generalized software as a program generator. (Source code can then be modified if necessary).
- Write the program generator in the COBOL 68-74 subset as specified in APPENDIX B.
- Design the program generator to write the generalized editor in the same COBOL 68-74 subset.
- Confine as many implementation dependencies as possible to the CONFIGURATION SECTION of the edit generator.
  e g file device names, date-register-name.
- Document clearly within the CONFIGURATION SECTION any other implementation dependencies required and where they occur in the source of the edit generator.
  e g the table containing the CONFIGURATION SECTION to be included in the generated editors.
- Use the long form of any reserved words.
  e g PICTURE not PIC, COMPUTATIONAL not COMP, etc.
- Limit numeric picture sizes to a maximum of twelve digits.
- Limit alphanumeric picture sizes to a maximum of 100 characters.
- Provide mechanism to cater for user exceptions.

6.3 Modularity

Modularity is a term used to describe the structure attribute of a program. A modular program has a top-down structure i e the complex tasks within the program are broken down into smaller sub-tasks (modules). In turn, each sub-task may be broken down into further sub-tasks. A sub-task is designed to be a logical unit. This technique is also referred to as structured programming.

There are many advantages of modular programming; some of them are listed below.

- The complexity of the program is decreased.
- Several programmers can code different modules for the same program simultaneously.
- Frequently used operations can be converted to a simple module which only needs to be written once.
- Debugging time is reduced because errors are generally isolated to a single module making it easier to find them.
- The program is easy to maintain; enhancements can be made with a better chance of avoiding widespread side effects.

6.3.1 The modularity of COBOL

COBOL is easily structured. Procedure statements are grouped together as a module called a paragraph. Paragraphs can be grouped together as a module called a section. The PERFORM statement can be used to evoke execution of one or more sections or one or more paragraphs. Repetitive execution of the same module is also possible with the PERFORM statement.

Several of the COBOL statements are themselves a module, i e they perform the task of several statements. The MOVE CORRESPONDING

statement is one. It operates as if it were several MOVE statements. The ADD CORRESPONDING and SUBTRACT CORRESPONDING are similar. If a file needs sorting, then many procedure statements could be written to perform this task or the SORT statement could be used. Similarly the COBOL report writer module can be used to simplify the task of generating reports.

6.3.2 The modularity of the generalized editor

The benefits of modular programming are important for any software development. Both the program generator and the generated editor should be modular in structure. This is particularly important for the editor because then its size can vary according to the complexity of the editing problem it is generated for. As an example, in an application where the input data to be validated is not batched, the editor need not contain all the code required for batch checking. (See Fig 6.2). Keeping the generated editors to their optional sizes is a desirable objective. They will run as efficiently as possible and require the minimum machine space.

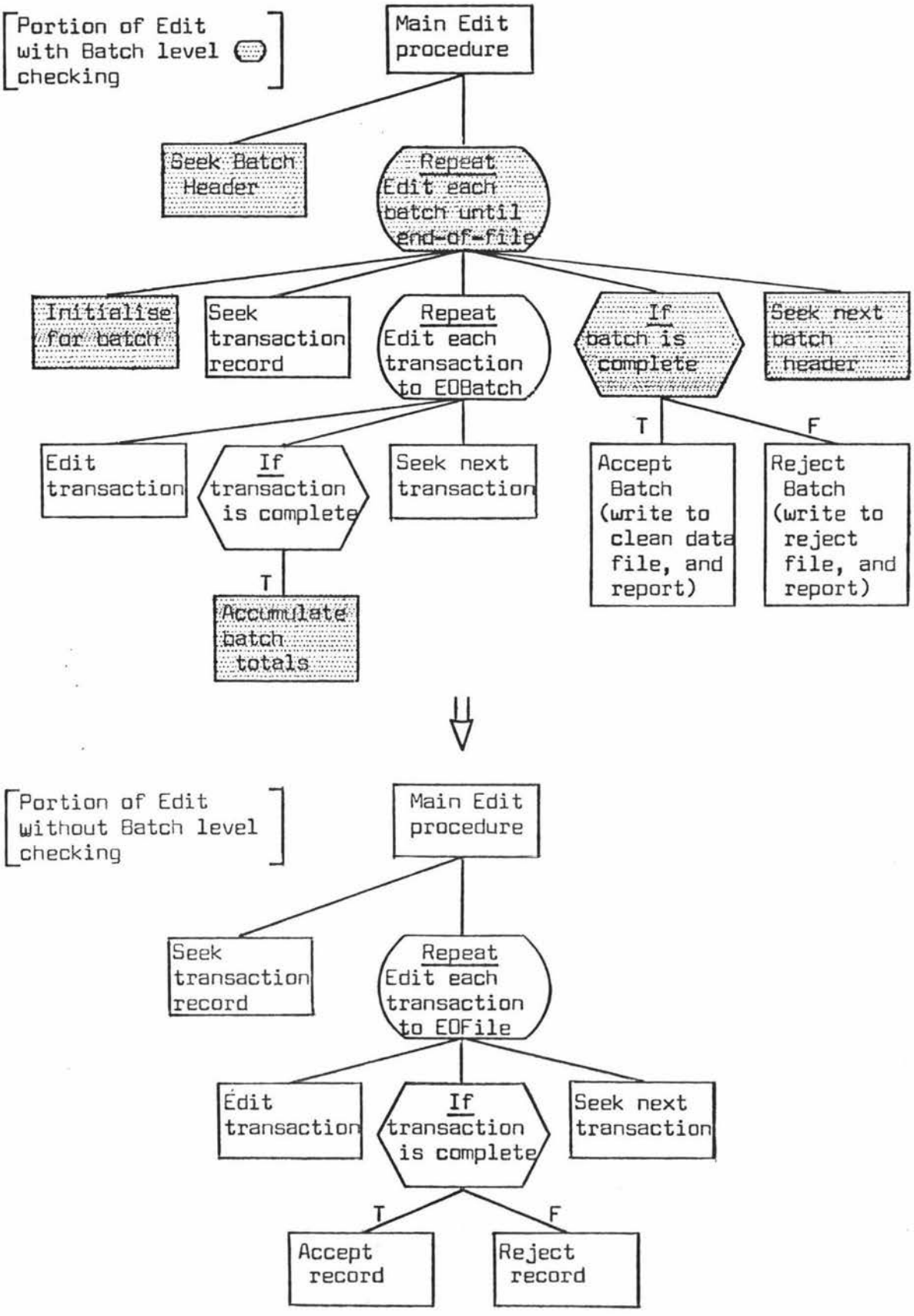


Fig 6.2 Modularity of the Editor.

6.4 Flexibility

If a program is structured to the extent that it can be modified and extended easily or it provides a high degree of generality in its functions, then it can be termed flexible. There are several reasons discussed in this section for the need of flexibility within the program generator and the generalized editor.

6.4.1 The flexibility of the generated editor

The generated editor needs to be flexible in its generality of editing tasks. These include the types of file environment, the types of error checks, and the types of reporting functions. User exception flexibility also needs to be provided. This will be possible by permitting the inclusion of user-written procedures.

In addition to providing the majority of editing functions and the mechanism of including user-written procedures, further software tailoring can be achieved. Since the editor is generated as source-code it is relatively easy for the user to make additional modifications to it, particularly if an on-line text editor is available.

6.4.2 The flexibility of the program generator

Experience in generalized software development shows that few initial implementations remain static. Typically, enhancements are made, redundancies are removed and extensions are provided to cater for unforeseen problems and requirements. Section 6.2.2 on the portability of COBOL, discusses the history of such a development, in a programming language.

As this implementation specification is an outline of the initial design of a generalized editor, it cannot be presumed to solve

the editing problem exhaustively. The implementation should be flexibly written with this in mind. An objective approach would be to implement in stages, starting first with a basic version and building on to it later as the new requirements become obvious. For example, the initial editor could provide for basic class checking of numeric, alphabetic, and alphanumeric, with more complex combinations of these provided for in a later version. Similarly the initial version could limit the recycling-errors task to input and output of the appropriate files only. Correcting the rejected transactions could be left to an independent program or could be achieved through the inclusion of a user procedure.

6.5 Ease of Use

It is important for generalized software to be usable. Acceptance of software depends more on how easy it is to use than on how efficiently its machine-code runs. This software attribute is to some extent subjective, as are others discussed earlier. Ease of use depends on the skill of the user and the complexity of his problem. Some definite techniques however can be used to assist him. The ease of use of the generalized editor has been a major consideration in its design.

6.5.1 Unconstrained use of the generated editor

Running the generated editor should require a minimum of decision making. Some initialization parameters may be needed, (e g date of run, job number, etc) but essentially the editor should run as is. A report produced with the editor to be used as an accompaniment, should contain information such as: the structure of the edit, the different levels of checks, detail on which checks are performed, (showing field sizes, number ranges etc), the names and types of files produced, and the types of reports it will generate. Error messages should be meaningful and helpful in locating errors. The text of these messages should be kept in a common table so that they can be modified if the user so wishes.

6.5.2 Unconstrained use of the program generator

The specifications for an editor that are input to the program generator need to be simple in construction. It is pointless if a user needs to expend more effort in the writing of edit specifications than if he were to write a comparable specific editor. For example, in some applications it may be easier to write specific

reporting code than to use COBOL report writer. One of the most effective methods to assist the user when specifying parameters, is to provide default values for all but the essential program variables. This means that few inputs to the program are required. Defaults for the file device mnenomics, the file attributes, the report formats, etc, are examples. The essential inputs such as record formats and field checks required could also have component option defaults.

Error diagnostics for the input specifications should be as meaningful and helpful as those for the input data to the editor.

7. DETAILED DESIGN METHOLOGY OF GENERALIZED EDITING

7.1 Design Introduction

This chapter reviews the functions and operations required of a generalized editor. It includes a discussion on the design of input specifications and how these will be translated into the source-code of the generated program. The overall structure of the generated editor is also outlined.

Since extensive program implementation has not accompanied this design, a general approach has been taken. The design methology intends to identify some of the problems and includes possible solutions where they are appropriate. Development of an overall solution is made to the extent that the problem is clearly feasible.

7.2 Generalized Edit Functions and Operations

This section gives general detail of the functions and operations to be included in a basic generalized editor. Part of its functioning is the ability to cater for additional operations or exceptions required by the user.

7.2.1 Brief list of edit operations

Following is a brief list of the file environment, the error checks, and the reports provided by the basic edit program generator.

(1) File environment

a. Input

- Unedited Input data file.
- Rejected transactions file, (from previous edit).

b. Output

- Clean data file.

- Rejected transactions file.
- Edit report.

(2) Types of error checks

a. Field level

- Class.
- Justification.
- Presence.
- Range.
- Date.
- Table membership.
- Check digit.

b. Transaction level

- Transaction construction.
- Field sequencing.
- Copresence.

c. Batch level

- Control totals.
- Hash totals.
- Document counts.

d. File level

- Master file record correspondence.

(3) Report functions

a. Transaction report

- All transactions.
- Rejected transactions only.

b. Clean data report

- File dump of clean data.

7.2.2 Exhaustive list of edit operations

The tables below specify each of the edit functions in more detail. Defaults and alternatives are given for each option. The defaults help to make the generalized software easy to use. The alternatives show how flexible specifying an editor can be. (US) indicates that the user can specify the entry.

Table 7.1 File Environment Detail

(US) - User Specified

File Name / Attribute	Default	Alternatives	Description
UNEDITED-TRANS			This file contains the primary input of unedited transactions.
Mode	Input	-	An On-line I/O mode will be left for an advanced implementation.
Device	Card Reader	(US)	Alternatives - Disc, Tape, Cassette, etc
Record length	-	-	Record description will determine this value.
TRANS-REJECTED-IN			This file contains the rejected transactions from the previous edit(s).
Mode	Input	-	
Device	Disc	(US)	Alternatives - Tape, Cassette, etc.
Record length	-	(US)	Same length as for the UNEDITED-TRANS file.
TRANS-REJECTED-OUT			This file contains the rejected transactions output from the edit.
Mode	Output	-	
Device	Disc	(US)	

File Name Attribute	Default	Alternatives	Description
Record length	-	(US)	Same length as for the UNEDITED-TRANS file.
CLEAN-TRANS			This file contains the edited transactions that are not rejected.
Mode	Output	-	
Device	Disc	(US)	
Record length	-	(US)	Same length as for the UNEDITED-TRANS file.
EDIT-REPORT			Listing of transactions and editing results.
Mode	Output		
Device	Printer	(US)	Alternatives - VDU, Disc etc.
Record length	132 chrs/line	(US)	Alternatives - 120, 80 for VDU.
Lines/Page	60	(US)	Alternatives - 45 if 6 lines/inch.
Contents	All Input	Errors only	Transaction with error message text.
		Clean transactions	Produced after input edit is complete CLEAN-TRANS file is re-opened and the contents printed. (This report can be printed in addition to the other alternatives).
		Transactions rejected	Produced after input edit is complete. TRANS-REJECTED-OUT file is re-opened and the contents printed. (This report can be printed in addition to the other alternatives).

Table 7.2 Types of error checks

Values in parentheses () are user inputs (see Section 7.3.3). Each alternative may also include (U) to indicate user defined exceptions.

Check type	Default	Alternatives	Description
Class	(ø) Any character-set	(X)Alphanumeric	0 to 9, A to Z, and ø (space)
		(A)Alphabetic	A to Z, leading or trailing ø
		(9)Numeric	0 to 9, leading or trailing (ø, +, -)
Justification	(ø,Y) Right if numeric	(L)Left, (N)No	
		(R)Right,(N)No	Although ø is an Alphanumeric, leading or trailing spaces will be significant in determining correct justification.
	(ø,Y) Left if Alphabetic	(R)Right,(N)No	
Presence	(ø,Y) Present	(N)Not Present	
Range	(ø,N) No Range check	(Y)Range check	Any '88 level' list of values, (i e ascending order).
		(D)Date Range check	Any '88 level' list of date values. '*' used in the list means 'Today's date'.
Date	(ø,N) No Date check	(Y)Date check	Forms permissible: DDMMYY or MMDDYY etc DD/MM/YY (DD,MM 1 or 2 digits) DD-MM-YY DD.MM.YY
Table membership	(ø,N) No Table membership	(Y)Tbl membership	Simple tables only of the form: 01 TABLE 02 ITEM OCCURS n TIMES PICTURE X(m). (name, n, m user specified).
Check digit	(ø,N) No check	(Y,E)Mod 11 check D	Mod 11, Numeric using right most digit, no non-numeric elements weighting factor (1,2,3,4,5...) other factors (US)
		(T)Mod 10 check D	weighting factor (...121212) other factors (US)
Field sequence	(ø,N) No check	(Y,+)Seq increasing	Increments other than 1 must be (US)
		(-)Seq decreasing	Increments other than 1 must be (US)

Check type	Default	Alternatives	Description
Copresence	No check	Copresence existence Copresence limits	The list of field names that must coexist must be (US). The list of field names with value ranges that are copresence limits must be (US). The ranges of cofields will be implicitly 'ANDed'.
Transaction construction	One physical record per logical record	More than one physical record per logical record	The (US) transaction structure; alternate record orders and no. number of occurrences possibly varying. e g Rec1 [Rec2] [Rec3 [Rec6] ... / Rec4 Rec5]
Batch totals	No Batch checks	Control totals Hash totals Document count	A list of field-names; each will be totaled. A list of field-names; each will be totaled. Single total.
Master file record correspondence	-	-	Implemented in future version.

Table 7.3 Report layouts

This layout can apply to all reports.

Line type / Item type	Default	Alternatives	Description
Report heading			The actual detail and format for any of the headings can be specified by the user.
Installation-ID	line(1) size(132) value(spaces)	line(US) size(US) value(US)	Any installation comment entry required.
System Identification	line(2) size(132) value(spaces)	line(US) size(US) value(US)	Any system identification comment entry required.
Program Identification	line(3) size(132) value(EPG EDITOR)	line(US) size(US) value(US)	Any program identification comment entry required.
History	line(4) size(32) value(RUN DATE dd mm yy TIME hrs:mins)	line(US) size(US) value(US)	Any history comment entry required. The date and time information may need to be initialized from parameters to the editor if the appropriate hardware registers are not accessible.
Page heading-1			
Title	size(60) value(spaces)	(US)	Any title comment entry required.
First key	size(30) value(first key)	(US)	This is helpful as a thumb index for large report listings.
Batch No.	size(2) value(m)	(US)	Current batch number being edited.
Date	size(8) value(dd/mm/yy)	(US)	
Page No.	size(3) value(n)	(US)	
Page heading-2			
Column headings	size(132) value(field-names)	(US)	The field-names supplied in the record-descriptions can be truncated if necessary then used as headings.

Line type / Item type	Default	Alternatives	Description
Batch heading			
Batch No.	size(2) value(m)	(US)	
No. transactions in batch	size(2) value(t)	(US)	
First & last keys	size(30x2) value(first & last key)	(US)	
Transaction detail			
Fields	size(as declared)	(US)	The fields to be selected for printing may be user specified. Normally a single ∅ (space) will separate each field.
(possible) Error messages	size(80) value(error text)	(US)	Item identification will be by naming or column position. It is not as practical to use position indicators since the field in error may not be listed.
Batch footing			
Batch No.	size(2) value(m)	(US)	
Pre-entered batch info.	size(as required)	(US)	
Computed batch info.	size(as required)	(US)	
Batch messages	size(as required)	(US)	
Page footing			
Batch No.	size(2) value(m)	(US)	
Last key	size(30) value(last key)	(US)	This is helpful as a thumb index for large report listings.
Report footing			
No. transactions	size/value(as required)	(US)	
No. rejections	"	(US)	
No. batches	"	(US)	

7.3 Input Specifications

The specifications to the edit generator need to be carefully designed. Being a man-machine interface they should be not only intelligible and unambiguous but also as succinct as possible. This section discusses the development towards a feasible solution to the input specification requirements.

7.3.1 The specifications that need to be communicated

In section 7.2 many of the functions and operations of the generalized editor were discussed. The detail in tables 7.1 to 7.3 show that many of these functions and operations are user specified.

Below is a list of the general specifications required.

(1) File environment

- Which files are required.
- File attributes, (device, title, blocking, etc).
- Record formats.

(2) Types of error checks to be performed

- Which checks.
- Appropriate check parameters.
- User exception actions.

(3) Report functions

- Which reports.
- Report contents.
- Report formats.

7.3.2 How to communicate specifications (development towards a feasible solution)

There are many methods possible for specifying the record descriptions

and the checks to be performed on each item. Some alternatives are discussed below. Included with each description is an example followed by a statement of its associated advantages or disadvantages. A feasible solution develops from the discussion.

(1) Specifying the record formats to uniquely identify items (fields).

a. Using standard COBOL record-descriptions:

(description)

When the user wishes to define a record format a standard COBOL record-description is supplied.

(example)

```
01  TRANSACTION-RECORD.
    02  CARD-CODE                  PICTURE 9.
    02  ORDER-DATE.
        03  DY                     PICTURE 99.
        03  MH                     PICTURE 99.
        03  YR                     PICTURE 99.

    02  ORDER-NUMBER               PICTURE X(5).
    02  QUANTITY-ORDERED           PICTURE 9(4).
    02  ITEM-NUMBER                PICTURE 9(5).
    02  ITEM-DESCRIPTION           PICTURE X(24).
    02  COST                       PICTURE 999V99.
```

(advantages)

- Often the user will be familiar with this COBOL notation.
- The user is specifying at a high level in free format. Each item is named and given a PICTURE class and size.
- Groups of items can be named.
- The record as a whole is named.
- Changing the record layout is easily performed by inserting new items, deleting items, or rearranging items.

- A copy of the record description is often available, (i e a COBOL Library file). Recoding of the record description is saved if the user can insert it from a Library.

(disadvantages)

- Extensive code will be necessary to scan the wide range of possible PICTURE string formats. This might make it difficult to run the program generator on smaller machines.

b. Table reference numbers, (i e a Data Dictionary [7]):

(description)

Each field within the record description has a table reference number. This number is an index into a subordinate file. The corresponding file entry will contain the item's name, class and size along with other detail.

(example)

Record	Field	Table reference	Description (Comment only)
1	1	077	CARD CODE
	2	043	ORDER DATE
	3	202	ORDER NUMBER
	4	203	QUANTITY ORDERED
	5	144	ITEM NUMBER
	6	146	ITEM DESCRIPTION
	7	058	COST

(advantages)

- The record is relatively quick to define.
- The table can be updated separately.

(disadvantages)

- Apart from the description entry it is difficult to recognise the

function of each item, (i e the scheme is not self documenting).

- It would be easy to specify an incorrect reference.
- The problem of defining the item has only been shifted. Defining the table entries is not necessary.
- Portability of the editor is decreased as to date (1978) not many installations in N Z have data dictionaries let alone a standardized format for them.
- Internal data names may need to be generated. These will make the editor source code less readable.

  e g The following record description could result.

```
01   RECORD-1.
     02   R1F1          PICTURE X(01).
     02   R1F2          PICTURE X(06).
     02   R1F3          PICTURE X(05).
     02   R1F4          PICTURE X(04).
     02   R1F5          PICTURE X(05).
     02   R1F6          PICTURE X(24).
     02   R1F7          PICTURE X(05).
```

- Similarly the PROCEDURE DIVISION code would be less readable.

  e g IF R1F7 IS LESS THAN ZERO ...

  is not as meaningful to the reader as

  IF COST IS LESS THAN ZERO ...

- The presence of the table is essential, thus making the edit generator less autonomous.

c. Column and Class indicators:

(description)

Each field is defined by a column-start and column-end indicator. Its class has to be defined by some code indicator.

(example)

Column Start	Column End	Class	Decimal places
1	1	N	-
2	7	N	-
8	12	X	-
13	16	N	-
17	21	N	-
46	50	N	2

(advantages)

- Unique naming is unnecessary.
- This scheme follows an RPG (Report Program Generator) format. (A widely used data processing language).

(disadvantages)

- As with the table reference scheme, internal names may need to be generated affecting source-code readability.
- The terseness of this method encourages specification errors.
- The scheme is not self documenting.
- Any changes to the record layout may necessitate extensive modifications to these specifications, (i e to insert or delete a field all subsequent column indicators need to be adjusted).

d. Field size and class indicators:

(description)

Instead of specifying column start and end, the size only is given.

The order of items defined is essential in defining the location of a field.

(example)

Size	Class	Decimal places
1	N	-
6	N	-
5	X	-
4	N	-
5	N	-
24	X	- (filler)
5	N	2

(advantages)

- This is a simple definition scheme.
- Rearranging the record layout is easy.

(disadvantages)

- Once more source-code readability is affected as naming is not supplied by the user.
- Fields not needing reference have to be defined so that the correct position of subsequent items is ensured, (i e some FILLER equivalent is needed).
- The scheme is not self documenting.

(2) Specifying the checks required for each item.

a. Using free format verbs:

(description)

Each check is named, e g PRESENT, NUMERIC, LEFTJUSTIFIED, RANGE(47 - 198), etc. A list of the checks required follows each field definition. If the fields are defined through the use of COBOL record-descriptions

then this list can be included as a comment either on an adjacent line or after the COBOL field description entries.

(example)

```
    CARD-CODE                PRESENT, NUMERIC, RANGE (6-6)
    ORDER-DATE               PRESENT, DATE, INCREASING (+1)
    ORDER-NUMBER
    QUANTITY-ORDERED         PRESENT, NUMERIC
    ITEM-NUMBER              PRESENT, NUMERIC, MEMBER (ITEM-TABLE),
                               CHECK-DIGIT
    ITEM-DESCRIPTION
    COST                     PRESENT, NUMERIC
```

(advantages)

- This high level method is self documenting.
- Only the required checks need to be specified.

(disadvantages)

- Specifying the checks can become verbose.
- Incorporating the checks with the COBOL record-descriptions means: a scanner is needed to differentiate check verbs from comment text or other language constructs, the readability of the COBOL program is deteriorated, record-descriptions kept in libraries may also be COBOL incompatible when used for separate applications.

e g (i) As comment entries

```
     01   TRANSACTION-RECORD.
          02   CARD-CODE          PICTURE 9.
     *              PRESENT, NUMERIC, RANGE (6-6)
          02   ORDER-DATE         PICTURE 9(6).
     *              PRESENT, DATE, INCREASING (+1).
```

(ii) After a special escape character

```
     01   TRANSACTION-RECORD.
          02   CARD-CODE    PICTURE 9.  % PRESENT, NUMERIC,
     -                                     RANGE (6-6)
          02   ORDER-DATE   PICTURE 9(6).  % PRESENT, DATE,
     -                                     INCREASING  (+1)
```

- The record-description will need to be redefined for class checking purposes. It is no good testing CARD-CODE for NUMERIC with a PICTURE 9.

  e g either

```
01   RECORD-1    REDEFINES TRANSACTION-RECORD.
     02  R1F1         PICTURE X(01).
     02  R1F2         PICTURE X(06).
                 etc
```

  or

```
01   RECORD-1    REDEFINES TRANSACTION-RECORD.
     02   R1-STG   OCCURS n TIMES PICTURE X(01).
```

b. Using a fixed format check-list table:

(description)

With each item defined for the transaction single-character indicators in set columns are used to specify which checks are required. Information such as range values and table names need to be specified as additional detail.

(example)

		Justification					Check
	Present	Left	Right	Range	Date	Seq	Digit
CARD-CODE	x			x			
(Range (6))							
ORDER-DATE	x				x	x	
(Seq(Increasing))							
ORDER-NUMBER			x				
QUANTITY-ORDERED	x						
ITEM-NUMBER	x						x
(Member (Item-table))							
ITEM-DESCRIPTION		x					
COST	x						

(advantages)

- All necessary information is present.
- Easy to interpret.
- Specifying the required checks is systematic.

(disadvantages)

- Double entering is needed for some checks, (e g for a range check an 'x' is entered in the RANGE column and the bound values are entered in the INFO field.

7.3.3 Feasible solution to the input specification requirements

As well as being feasible for implementation the input specifications discussed below are designed to maximize the achievement of the generalized software objectives, (i e ease of use, flexibility, and self documentation).

The majority of users likely to require generalized editing will be familiar with the COBOL language. Because of this the COBOL coding format has been chosen for input specification use.

i e	Columns	Function
	1-6	Sequence (Not used)
	7	Continuation or Comment indicator
	8-11	Start of headings
	12-72	Edit detail
	73-80	Identification (Not used)

(See Fig 7.4)

Many of the specifications will be similar to the COBOL language in structure. They will appear in a DIVISION and SECTION order. An almost direct translation to COBOL code will be possible. Most DIVISION and SECTION options not specified by the user will have

COBOL Coding Form

SYSTEM		PUNCHING INSTRUCTIONS		PAGE OF
PROGRAM		GRAPHIC	CARD	IDENTIFICATION 73 80
PROGRAMMER	DATE	PUNCH	FORM #	

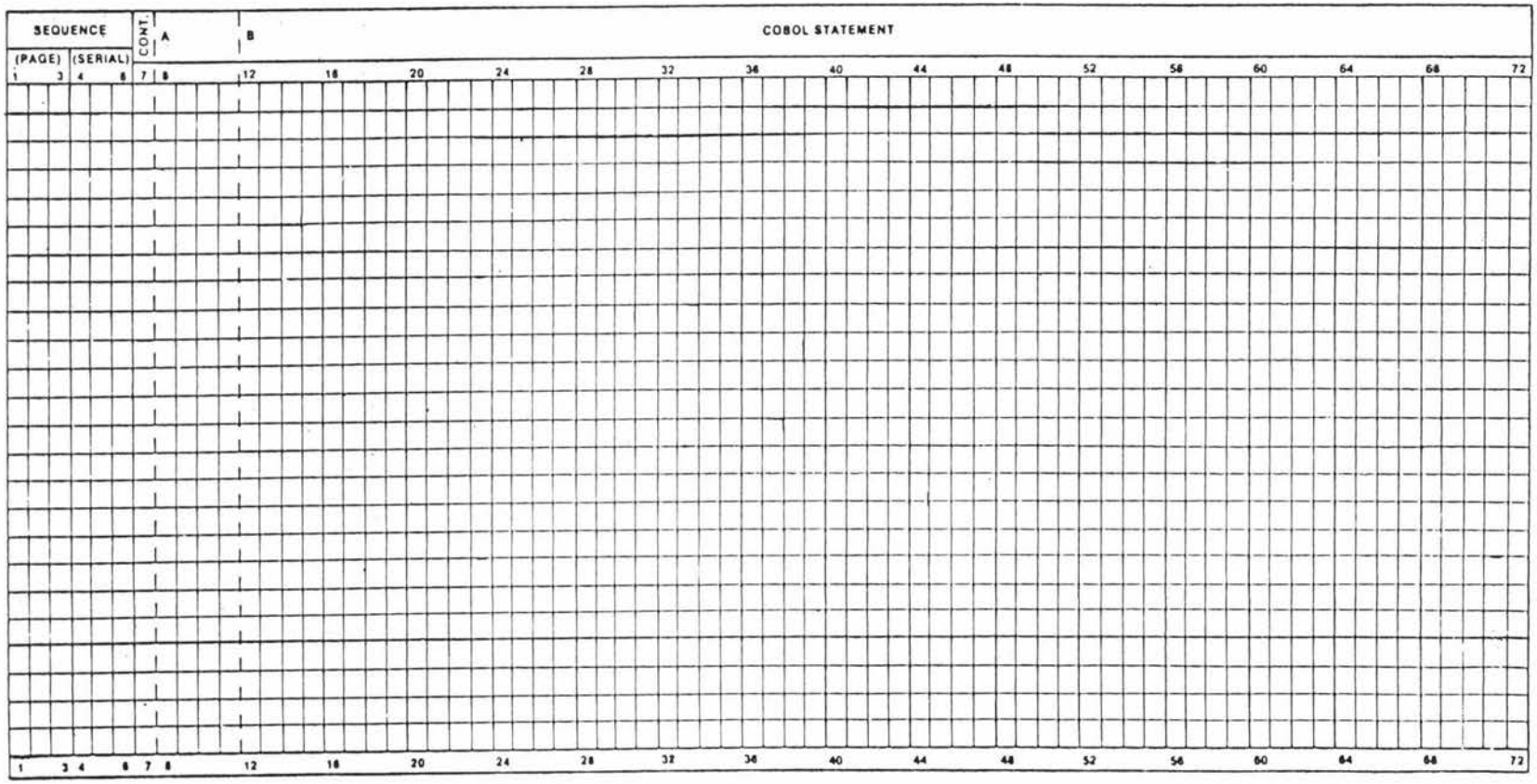


Fig 7.4 Sample COBOL coding form, used for the edit specifications.

default values. This means that the user will be able to include as much or as little of the program detail as required.

Generally a fixed format approach will be chosen for those edit specifications that are essential input. For example, the method of defining each field identifier and the item checks to be applied, will be as follows:

(1) Field-name - Column 12-41

This must be left justified, (i e the first letter must be in Column 12). All the conventions for a COBOL name apply, (i e not more than 30 characters from A-Z, 0-9, or - (hyphen), and a word may not begin or end with a hyphen).

(2) Class - Column 42 containing (Ø, 9, A, X, or U).

(3) Size - Column 44-48 containing (integer, digits V digits to indicate decimal places for Class '9' items, or U).

(4) Presence - Column 50 containing (Ø, Y, N, U).

(5) Justification - Column 52 containing (Ø, Y, L, R, N, U).

(6) Range - Column 54 containing (Ø, N, Y, D, U).

(7) Date - Column 56 containing (Ø, N, Y, U).

(8) Table Membership - Column 58 containing (Ø, N, Y, U).

(9) Check digit - Column 60 containing (Ø, N, Y, E, T, U).

(10) Sequence - Column 62 containing (Ø, N, Y, +, -, U).

(11) User check - Column 64 containing (Ø, N, Y, U). Y, U are synonyms.

(12) Info - Column 66 - 72 or Column 16 - 72.

These fields are used to specify any additional detail. If 'U' is specified in any of the check fields then additional detail

must be present. The detail will be presented in parentheses following a type name. A list of such detail may be specified. (e g PRESENT (User-procedure-name), RANGE (7-143, 206), SEQUENCE (+2)).

See Fig 7.5 for a record description example.

Other input detail is specified in a similar combination of fixed and free format. More information on these various formats is given in the next section.

COBOL Coding Form

SYSTEM	PUNCHING INSTRUCTIONS		PAGE OF
PROGRAM	GRAPHIC	CARD FORM #	IDENTIFICATION 73–80
PROGRAMMER — DATE	PUNCH		

SEQUENCE (PAGE) (SERIAL) — CONT. — A — B — COBOL STATEMENT

(Presence) (Justification) (Range) (Date) (Membership) (Check Digit) (Sequence) (User Procedure)

```
*
*  SAMPLE RECORD-DESCRIPTION EDIT SPECS.
*
```

Field-name	Class	Size	P	J	R	D	M	C	S	U	Info.
TRANSACTION-RECORD											
CARD-CODE	9	2	Y	Y	Y	N	N	N	N	N	R(6)
ORDER-DATE	9	6				Y			+		
ORDER-NUMBER	X	5	N	R							
QUANTITY-ORDERED	9	4									
ITEM-NUMBER	9	5					Y	E			
– MEMBER(ITEM-TABLE)											
ITEM-DESCRIPTION	X	24	N	L							
COST	9	3V2									
FILLER		10									

Fig 7.5 Sample record-description edit specifications.

7.4 Conversion of Input Specifications to Source-code

In this section the manner in which the input specifications may be translated to COBOL source-code is out-lined. This helps to show the relationship between the input specifications and the generated COBOL program. Many of the input specifications definitions have been left until this section in the interests of conciseness.

7.4.1 Using the COBOL language format.

The three general specification needs of: file environment, error checks to be performed, and report functions, as listed in section 7.3.1, can be positioned logically into a COBOL language format. For example, the file environment specifications can be FILE SECTION entries, and the error checks to be performed can be WORKING-STORAGE SECTION entries. This approach permits a high degree of program flexibility. Additional program segments can be easily included at will by the user. In the situations where sections of the COBOL program to be generated are not supplied by the user, default code can be substituted. For example, it the user does not supply any IDENTIFICATION DIVISION detail the following skeleton will be generated.

```
IDENTIFICATION DIVISION.
        PROGRAM-ID.             GENERALIZED EDITOR.
        AUTHOR.                 EPG (EDIT PROGRAM GENERATOR).
        INSTALLATION.           *NOT SPECIFIED*.
        DATE-WRITTEN.           date.
        DATE-COMPILED.
        SECURITY.               *NOT SPECIFIED*.
```

Note. These entries are copied from a table within the program generator and can be initialized to any particular installation requirements.

If the user wishes to deviate from the use of this default DIVISION format then the following could be specified.

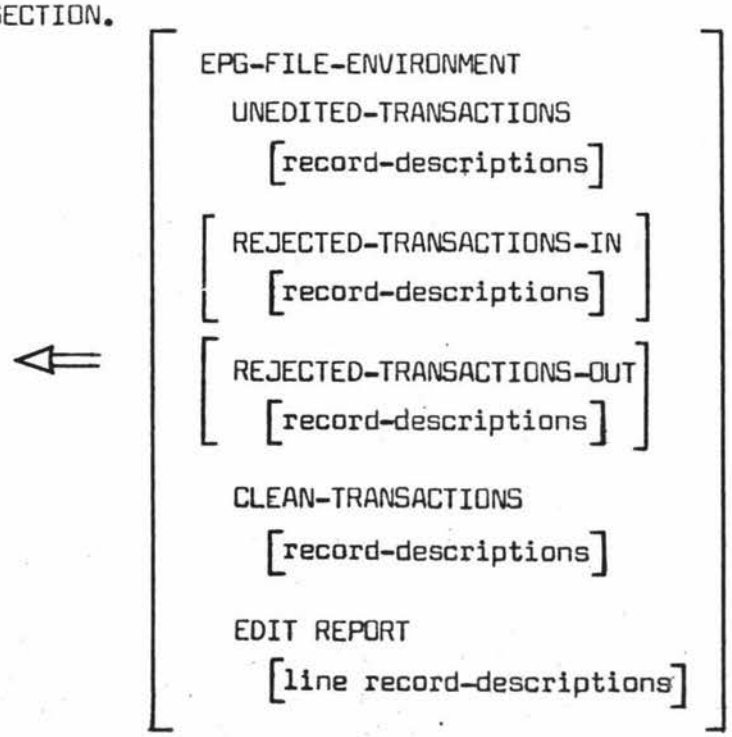
```
EPG-IDENTIFICATION DIVISION.
  PROGRAM-ID.          EDITOR FOR SALES ORDERS.
  AUTHOR.              EPG.
  INSTALLATION.        NORTHERN GAT-LING COMPANY.
  DATE-WRITTEN.        CIRCA 1861.
  SECURITY.            NONE.
```

Note. The significance of the EPG prefix to the DIVISION name is explained in the next section.

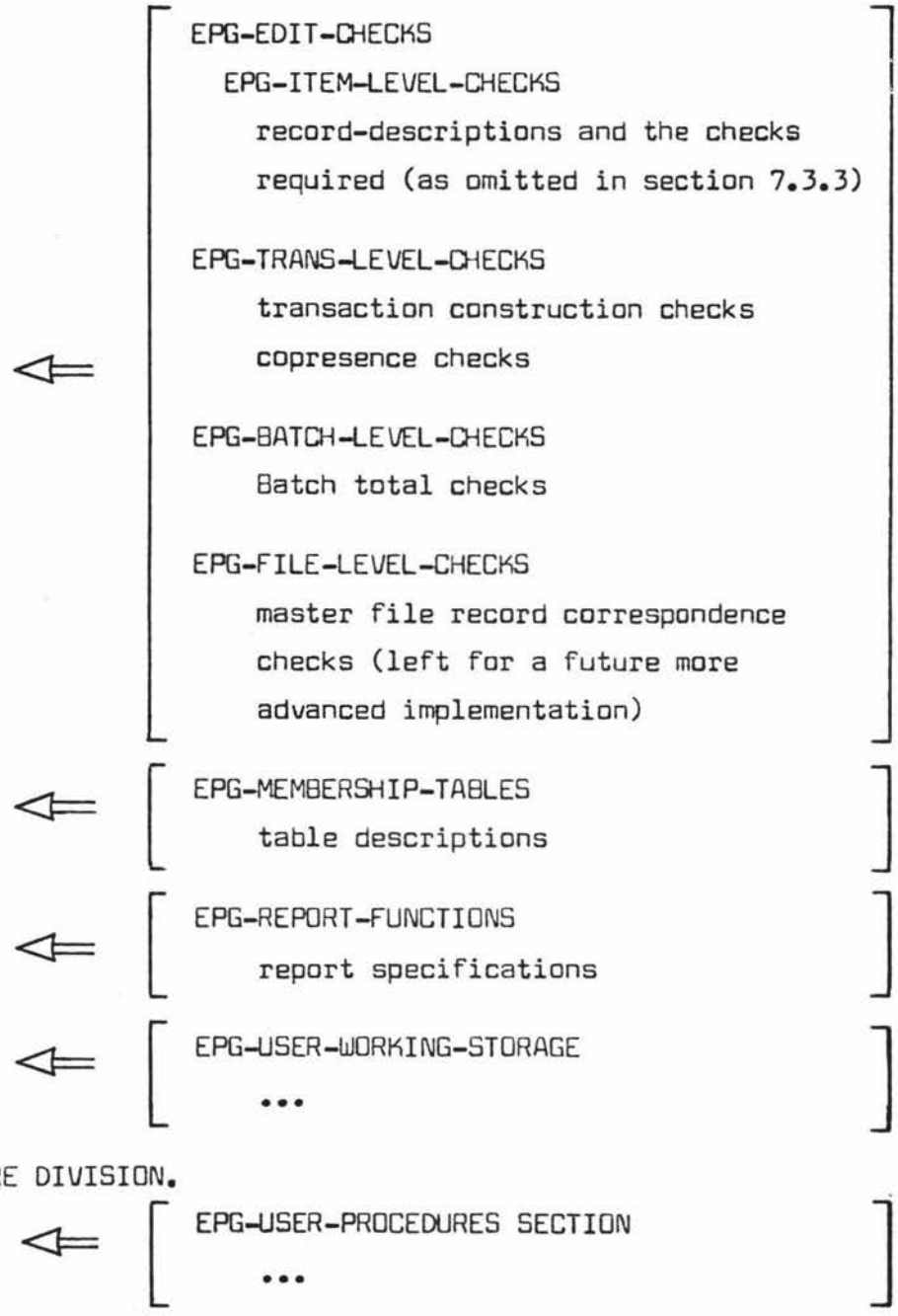
Shown below is a COBOL program skeleton relating user specifications to the four DIVISIONS.

COBOL Skeleton	User specifications

```
IDENTIFICATION DIVISION.
     ...
ENVIRONMENT DIVISION.
     ...
DATA DIVISION.
  FILE SECTION.
```



WORKING-STORAGE SECTION.



PROCEDURE DIVISION.

7.4.2 Naming conventions

A naming notation is necessary to resolve any accidental naming conflicts with user inserted program segments. It is also necessary to assist the program generator when parsing of input specifications.

The edit specification headings are prefixed by the string 'EPG-'. This means the user is free to use any names except those commencing with this string. Such a convention is simple to adopt. Generated paragraph-names, file-names, etc, can all be prefixed this way.

It is assumed that record-names specified by the user are unique. If however there is a need to generate a name based on a user supplied option then uniqueness must be ensured. Simply prefixing the users' name by 'EPG-' may cause problems if the resulting name then exceeds 30 characters in length, (the maximum allowed for COBOL names). One solution is to use a unique four digit integer between 'EPG-' and a truncated portion of the users' name, e g EPG-0043-REGISTRATION-NUMBER-T. Note. Care must be taken that the truncated name does not end with a hyphen.

Following is a list of special names known to the program generator. If they are used they must start in margin A of the coding form, (i e columns 8 - 11).

Name	Abreviation
EPG-IDENTIFICATION DIVISION	EPG-ID
EPG-ENVIRONMENT DIVISION	EPG-ED
EPG-DATA DIVISION	EPG-DD
EPG-FILE SECTION	EPG-FS
EPG-FILE ENVIRONMENT	EPG-FE
EPG-WORKING-STORAGE SECTION	EPG-WS
*EPG-EDIT-CHECKS	EPG-EC
*EPG-ITEM-LEVEL-CHECKS	EPG-IC

EPG-TRANS-LEVEL-CHECKS	EPG-TC
EPG-BATCH-LEVEL-CHECKS	EPG-BC
EPG-FILE-LEVEL-CHECKS	EPG-FL
EPG-MEMBERSHIP-TABLES	EPG-MT
*EPG-REPORT-DESIGN	EPG-RD
EPG-USER-WORKING-STORAGE	EPG-US
EPG-PROCEDURE-DIVISION	EPG-PD
EPG-USER-PROCEDURES	EPG-UP
EPG-BEGIN-EXTRA-CODE	EPG-BEGIN
EPG-END-EXTRA-CODE	EPG-END

* These headings identify program specification sections mandatory to the generator input. Other names are used to correctly locate optional program detail.

7.4.3 The translation of specifications to source-code in detail

Detail on the translation to source-code from input specifications is discussed in this section. The code produced from default and user options is shown.

(1) IDENTIFICATION DIVISION

(default code)

```
IDENTIFICATION DIVISION.
  PROGRAM-ID.      GENERALIZED EDITOR.
  AUTHOR.          EPG (EDIT PROGRAM GENERATOR).
  INSTALLATION.    *NOT SPECIFIED*
  DATE-WRITTEN.    date.
  DATE-COMPILED.
  SECURITY.        *NOT SPECIFIED*
```

(input specification)

```
EPG-IDENTIFICATION DIVISION.
    user entries up to next EPG heading.
```

(code produced)

```
IDENTIFICATION DIVISION.
     user entries
```

(2) ENVIRONMENT DIVISION

(default code)

```
ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
  SOURCE-COMPUTER.      installation   entry
  OBJECT-COMPUTER.      installation   entry
  SPECIAL-NAMES.
        installation entry  IS CARD-READER
        installation entry  IS LINE-PRINTER
        installation entry  IS DISC
        installation entry  IS TAPE.

 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
        SELECT EPG-UNEDITED-TRANS     ASSIGN TO CARD-READER.
        SELECT EPG-CLEAN-TRANS        ASSIGN TO DISC.
        SELECT EPG-EDIT-REPORT        ASSIGN TO LINE-PRINTER.
```

(input specification)

```
EPG-ENVIRONMENT DIVISION.
      user entries up to next EPG heading
```

(code produced)

```
ENVIRONMENT DIVISION.
      user entries
```

(3) DATA DIVISION

a. FILE SECTION

(default code)

```
DATA DIVISION.
 FILE SECTION.
  FD    EPG-UNEDITED-TRANS.
     (user defined record-descriptions from EPG-ITEM-LEVEL-CHECKS
      section)
  FD    EPG-CLEAN-TRANS.
      (same record-descriptions as for EPG-UNEDITED-TRANS file)
  FD    EPG-EDIT-REPORT.
  01    EPG-LINE-BUFF        PICTURE X(132).
```

(input specification) (example)

```
EPG-FILE-ENVIRONMENT
 TRANSACTIONS-REJECTED-OUT                    } file name
  TRANSACTION-RECORD                          } record name
   CARD-CODE                    2             }
   FILLER                      10             }
   ORDER-DATE                   6             }
   ORDER-NUMBER                 5             } record description
   QUANTITY-ORDERED             4             }
   ITEM-NUMBER                  5             }
   COST                         5             }

TRANSACTIONS-REJECTED-IN
```

(code produced)

(i) appended to the I/O SECTION. (example)

```
    SELECT EPG-TRANS-REJECTED-IN ASSIGN TO DISC.
    SELECT EPG-TRANS-REJECTED-OUT ASSIGN TO DISC.
```

(ii) appended to the FILE SECTION. (example)

```
FD    EPG-TRANS-REJECTED-IN.

01    EPG-TRI-TRANSACTION-RECORD.
      02   CARD-CODE            PICTURE X(02).
      02   FILLER               PICTURE X(10).
      02   ORDER-DATE           PICTURE X(06).
      02   ORDER-NUMBER         PICTURE X(05).
      02   QUANTITY-ORDERED     PICTURE X(04).
      02   ITEM-NUMBER          PICTURE X(05).
      02   COST                 PICTURE X(05).

FD    EPG-TRANS-REJECTED-OUT.

01    EPG-TRO-TRANSACTION-RECORD.
      02   CARD-CODE            PICTURE X(02).
      02   FILLER               PICTURE X(10).
      02   ORDER-DATE           PICTURE X(06).
      02   ORDER-NUMBER         PICTURE X(05).
      02   QUANTITY-ORDERED     PICTURE X(04).
      02   ITEM-NUMBER          PICTURE X(05).
      02   COST                 PICTURE X(05).
```

(iii) appended to WORKING-STORAGE SECTION.

```
        any working variables needed.
```

(iv) appended to PROCEDURE DIVISION. (example)

```
      OPEN INPUT EPG-TRANS-REJECTED-IN.
      OPEN OUTPUT EPG-TRANS-REJECTED-OUT.
      ...

      READ EPG-TRANS-REJECTED-IN
           AT END MOVE "T" TO EPG-EOF-TRI
                  GO TO EXIT-GET-TRI-RECORD.

      MOVE CORRESPONDING EPG-TRI-TRANSACTION-RECORD
           TO TRANSACTION-RECORD.
      ...
```

```
MOVE CORRESPONDING TRANSACTION-RECORD
     TO EPG-TRO-TRANSACTION-RECORD.

WRITE EPG-TRO-TRANSACTION-RECORD
     INVALID MOVE "T" TO EPG-EOF-TRO
             GO TO EXIT-OUT-TRO-RECORD.
...

CLOSE EPG-TRANS-REJECTED-IN.
CLOSE EPG-TRANS-REJECTED-OUT.
```

b. WORKING-STORAGE SECTION

- Item level checks

(default code)

none

(input specification) (example)

```
  EPG-ITEM-LEVEL
   TRANSACTION-RECORD                        p j r d m c s u info.
    CARD-CODE                      9     2     Y             R(6)
    ORDER-DATE                     9     6        Y     +
    ORDER-NUMBER                   X     5 N R
    QUANTITY-ORDERED               9     4
    ITEM-NUMBER                    9     5           Y E
-       MEMBER(ITEM-TABLE)
    ITEM-DESCRIPTION               X    24 N L
    COST                           9   3V2
    FILLER                            10
```

check	default(ø)	alternatives	
Class		9, A, X, U	
Size	-	integer, digits V digits, U	
Presence	Y	N, U	
Justified	L, Y	R, N, U	(Alphanumeric)
Justified	R, Y	L, N, U	(Numeric)
Range	N	Y, D, U	(D - Date range)
Date	N	Y, U	
Membership	N	Y, U	
Check-digit	N	Y, E, T, U	(E - Mod 11, T - Mod 10)
Sequence	N	Y, +, -, U	
User check	N	Y, U	(Y, U are synonyms)

Item Information Codes (up-n - user-procedure-name)

CLASS or CL (up-n)
SIZE or SZ (value)
PRESENT or P (up-n)
JUSTIFIED or J (up-n)
RANGE or R (list of values)
RANGE or R (up-n) (if 'U' is specified)
DATE or D (up-n)
MEMBER or M (table-name)
MEMBER or M (up-n) (if 'U' is specified)
CHECK-DIGIT or C (formula) (default MOD 11)
CHECK-DIGIT or C (up-n) (if 'U' is specified)
SEQUENCE or S (integer)
SEQUENCE or S (up-n) (if 'U' is specified)
USER-CHECK or U (up-n)

(i) (code produced - in WORKING-STORAGE SECTION) (example)

```
WORKING-STORAGE SECTION.
 01    TRANSACTION-RECORD.
       02   CARD-CODE                 PICTURE X(02).
            88   EPG-RANGE-CARD-CODE VALUE "6".
       02   ORDER-DATE                PICTURE X(06).
       02   ORDER-NUMBER              PICTURE X(05).
       02   QUANTITY-ORDERED          PICTURE X(04).
       02   ITEM-NUMBER               PICTURE X(05).
       02   ITEM-DESCRIPTION          PICTURE X(24).
       02   COST                      PICTURE X(05).
       02   FILLER                    PICTURE X(10).

(plus other working variables required by the checking routines).
```

(ii) in FILE SECTION

```
FD    EPG-UNEDITED-TRANS.

01    EPG-UP-TRANSACTION-RECORD.
      02   CARD-CODE                PICTURE X(02).
      02   ORDER-DATE               PICTURE X(06).
                etc

FD    EPG-CLEAN-TRANS.

01    EPG-CT-TRANSACTION-RECORD.
      02   CARD-CODE                PICTURE X(02).
      02   ORDER-DATE               PICTURE X(06).
                etc
```

(iii) in record section of WORKING-STORAGE SECTION.

```
01    EPG-DETAIL-LINE-1.
      02   CARD-CODE                PICTURE X(02).
      02   FILLER                   PICTURE X(01) VALUE SPACE.
      02   ORDER-DATE               PICTURE X(06).
      02   FILLER                   PICTURE X(01) VALUE SPACE.
                etc
```

(iv) in PROCEDURE DIVISION

```
      record MOVE CORRESPONDING sentences.
      class checks, presence checks etc.
```

Checking code will be generated for each item.

e g

```
*TRANSACTION-RECORD ITEM CHECKS
* CARD-CODE
    EPG-R1F1-CHECK.
         class-check code
         present-check code
         range-check code

* ORDER-DATE
    EPG-R1F2-CHECK.
         class-check code
         present-check code
         data-check code
         sequence-check code

* ORDER-NUMBER
    EPG-R1F3-CHECK.
         class-check code
         justified-check code
   etc
```

Following are PROCEDURE DIVISION segments for some of the item-level checks. They are examples only. Implemented code may be different.

(Class numeric)

```
        IF item-name IN record-name NOT NUMERIC
             MOVE NOT-NUM-ERR TO ERROR-CODE
             PERFORM EPG-ERROR-ACTION
             GO TO EPG-RnIm-EXIT.
```

Note. The 'GO TO' is necessary to prevent any further checks from being made on the item. Multiple messages for the same item should be avoided, e g if an item value is the wrong class then it is also likely to be out of range.

The checking order should be as follows:

1st - present,
2nd - class,
3rd - justified,
4th - range, date, membership, check-digit,
5th - sequence,
6th - user procedure.

(Presence)

```
IF item-name IN record-name = SPACES
    MOVE NOT-PRSNT-ERR TO ERROR-CODE
    PERFORM EPG-ERROR-ACTION
    GO TO EPG-RnIm-EXIT.
```

Note. A numeric field of all 0's is assumed to be 'present'.

(Left justified)

```
IF item-name IN record-name NOT = SPACES
    IF LEFT CHAR OF item-name IN record-name = SPACE
        MOVE NOT-LT-ERR TO ERROR-CODE
        PERFORM EPG-ERROR-ACTION
        GO TO EPG-RnIm-EXIT.
```

Note. 1. This code is sufficient for numeric fields also.
e g If a numeric field should be left justified then ⍽040 will be detected as erroneous.

2. Those fields requiring justification checking, need to have a REDEFINES to declare LEFT-CHAR or RIGHT-CHAR.

```
e g   02  ORDER-NUMBER          PICTURE X(05).
      02  EPG-RD1 REDEFINES ORDER-NUMBER.
          03  RIGHT-CHAR        PICTURE X(1).
          03  FILLER            PICTURE X(04).
```

(Range)

```
IF NOT condition-name
     MOVE RANGE-ERR TO ERROR-CODE
     PERFORM EPG-ERROR-ACTION
     GO TO EPG-RnFm-EXIT.
```

Note. 1. The range check code needs to be more elaborate for date values.

2. If ranges are to be changed dynamically then range tables will need to be kept. This will be less efficient as table-search code must then be generated.

(Date)

More extensive code is necessary for performing date checks. A generalized date-check routine could be used.

e g

```
77   EPG-INVALID-DATE-FLAG    PICTURE 9.
     88   EPG-INVALID-DATE VALUE 1.

01   EPG-DATE-BUFF.
     02   EPG-DY                    PICTURE 99.
     02   EPG-MH                    PICTURE 99.
          88   EPG-VALID-MONTH VALUE 1 THROUGH 12.
     02   EPG-YR                    PICTURE 99.
...

     MOVE item-name TO EPG-DATE-BUFF.
     PERFORM EPG-CHECK-DATE.
     IF EPG-INVALID-DATE
          MOVE EPG-DATE-ERR TO ERROR-CODE
          PERFORM EPG-ERROR-ACTION
          GO TO EPG-RnFm-EXIT.
```

Note. More complex date formats may need to be checked.

e g DD/MM/YY, MM/DD/YY etc

Those could be left for the user to control in the initial implementation.

(Sequence)

```
01    EPG-SEQUENCE-CHECK-ITEMS.
      02   EPG-PREV-ORDER-DATE      PICTURE X(06).
...

      IF ORDER-DATE IN TRANSACTION-RECORD LESS THAN EPG-PREV-ORDER-DATE
           MOVE EPG-SEQ-ERR TO ERROR-CODE
           PERFORM EPG-ERROR-ACTION
           GO TO EPG-RnFm-EXIT
      ELSE MOVE ORDER-DATE IN TRANSACTION-RECORD TO
                                          EPG-PREV-ORDER-DATE.
```

Note. ORDER-DATE would have to be reordered YYMMDD before this comparison is possible.

- Transaction level checks

(default code)

none

(input specification) (example)

```
EPG-TRANS-LEVEL-CHECKS
 TRANSACTION-CONSTRUCTION
   STUDENT-ID-NAME RECORD-TYPE=I
   STUDENT-ADDRESS RECORD-TYPE=A
   (STUDENT-STATISTICS RECORD-TYPE=S) 2
   (FOREIGN-STUDENT S-NATIONALITY=F) 1-4
   (STUDENT-COURSES RECORD-TYPE=C) 1-8
   (STUDENT-RECORDS RECORD-TYPE=R) 0-*
   /EXTRAMURAL-RECORD RECORD-TYPE=E/
   /INTERNAL-RECORD RECORD-TYPE=I/
```

Note. The format of each line is:record-name item-name=value

If the record is optional then it is enclosed in parentheses.

Alternatives are enclosed by slashes.

Any entry may be post-fixed by a number range.

2 means exactly two records must be present.

1-4 means between one and four records must be present.

0-* or * means any number of records may be present.

Blank means exactly one record must be present.

(code produced)

In the PROCEDURE DIVISION,code will be generated to control the input and checking of each record composing the transaction.

(Input specification) (example)

```
COPRESENCE-EXISTENCE
  F1 AND F2 OR (F3 AND NOT F4)
COPORESENCE-LIMITS
  F1(4) AND F2(1,3,5)
```

(Code produced)

These copresence conditions will be converted to PROCEDURE DIVISION statements.

e g

```
    IF F1 NOT = SPACES AND F2 NOT = SPACES
        OR (F3 NOT = SPACES AND NOT F4 NOT = SPACES)
            call fault handling procedure
    IF F1 NOT = 4 AND F2 NOT = 1 OR 3 OR 5
            call fault handling procedure
```

- Batch level checks

(default code)

none

(input specification) (example)

	Class	Size	p	j	r	d	m	c	s	u	Information
EPG-BATCH-LEVEL-CHECKS											
EPG-BATCH-HEADER											
BATCH-NO	9	2	Y						+		B(BN)
BATCH-DATE	9	6	N		Y						
DEPT-FROM	X	20	N								
DEPT-TO	X	20	N								
FIRST-ORDER-NUMBER	X	6	Y								B(FK)
LAST-ORDER-NUMBER	X	5	Y								B(LK)
DOCUMENT-COUNT	9	2	Y		Y						B(DC)
R(25-55)											
QUANTITY-ORDERED	9	6	Y								B(CT)
COST	9	6V2	Y								B(CT)
ITEM-NUMBER	9	8	Y								B(HT)
EPG-BATCH-TRAILER											
etc											

Batch Information Codes

B(BN) - Batch number

B(DC) - Document count

BC(FK) - First control key

B(LK) - Last control key

B(CT) - Control total

B(HT) - Hash total

} item names declared here must correspond to identical names declared for the transactions

(i) WORKING-STORAGE SECTION (example)

```
01    EPG-BATCH-HEADER
      02    BATCH-NO                  PICTURE X(02).
      02    BATCH-DATE                PICTURE X(06).
      02    DEPT-FROM                 PICTURE X(20).
      02    DEPT-TO                   PICTURE X(20).
      02    FIRST-ORDER-NUMBER        PICTURE X(06).
      02    LAST-ORDER-NUMBER         PICTURE X(06).
      02    DOCUMENT-COUNT            PICTURE X(02).
      02    QUANTITY-ORDERED          PICTURE X(06).
      02    COST                      PICTURE X(08).
      02    ITEM-NUMBER               PICTURE X(08).

01    EPG-BATCH-VARIABLES.
      02    FIRST-ORDER-NUMBER        PICTURE X(06).
      02    LAST-ORDER-NUMBER         PICTURE X(06).
      02    DOCUMENT-COUNT            PICTURE 9(02).
      02    QUANTITY-ORDERED          PICTURE 9(06).
      02    COST                      PICTURE 9(08).
      02    ITEM-NUMBER               PICTURE 9(08).
```

The batch variables are accumulators used for batch comparison checks.

(ii) PROCEDURE DIVISION

Batch checking and rejection control code will be produced.

Checks on the batch header will also be coded.

- Table membership tables

(default code)

none

(input specification) (example)

```
EPG-MEMBERSHIP-TABLES
 ITEM-TABLE
   42701 42702 42703 42801 42803
   42809 42901
```

(code produced)

(i) WORKING-STORAGE SECTION

```
01    ITEM-TABLE.
      02   FILLER    PICTURE X(05)  VALUE 42701.
      02   FILLER    PICTURE X(05)  VALUE 42702.
      02   FILLER    PICTURE X(05)  VALUE 42703.
      02   FILLER    PICTURE X(05)  VALUE 42801.
      02   FILLER    PICTURE X(05)  VALUE 42803.
      02   FILLER    PICTURE X(05)  VALUE 42809.
      02   FILLER    PICTURE X(05)  VALUE 42901.
01    ITEM-ARRAY REDEFINES ITEM-TABLE.
      02   ITEM-MEMBER OCCURS 7 PICTURE 9(05).
```

(ii) PROCEDURE DIVISION

Table search routines for table membership item-level checks will be generated.

- Report functions

(default code)

Code will be produced to generate a report as outlined in Table 7.3 of section 7.2.2.

(input specifications)

```
EPG-REPORT-DESIGN
 REPORT-TYPE
   ALL-TRANSACTIONS
 REPORT-HEADING
   NEWPAGE
   NEWLINE
     20     "S T O C K   C O N T R O L   S Y S T E M"
   NEWLINE
     20     "= = = = =   = = = = = = =   = = = = = ="
   NEWLINES(2)
     27     "EDITOR FOR SALES ORDERS"
   NEWLINES(2)
     28   8 R-DATE
      3   8 R-TIME
   NEWLINES(2)
 PAGE-HEADING
   NEWPAGE
         2 BATCH-NO
     18     "SALES & ORDERS EDIT"
     11   5 ORDER-NUMBER
      3     "PAGE"
         4 PAGE-NO
   NEWLINES(2)
           "C O-DATE O-NO. QNTY I-NO. ITEM-DESCRIPTION
-          "     COST"
   NEWLINE
 BATCH-HEADING
       70 ALL "*"
```

```
'
NEWLINE
   2     "BATCH NO."
         4 BATCH-NO
   5     "DEPT FROM"
        15 DEPT-FROM
           "NUMBERED FROM  "
         5 FIRST-ORDER-NUMBER
NEWLINE
   2     "DATE "
         8 BATCH-DATE
  10     "TO   "
        15 DEPT-TO
   9     "TO    "
         5 LAST-ORDER-NUMBER
NEWLINES(2)
   2     "DOCUMENTS"
         4 DOCUMENT-COUNT
   5     "CONTROL-TOTALS"
  11     "HASH-TOTALS"
NEWLINE
  20   5 QUANTITY-ORDERED
   1     "QUANTITY-ORDERED"
   3   5 ITEM-NUMBER
   1     "ITEM-NUMBER"
NEWLINE
  20   5 COST
   1     "COST"
NEWLINES(2)
         "C O-DATE O-NO. QNTY I-NO. ITEM-DESCRIPTION
-        "     COST"
NEWLINE
DETAIL
NEWLINE
         1 CARD-CODE
   1     6 ORDER-DATE
   1     5 ORDER-NUMBER
   1     4 QUANTITY-ORDERED
   1     5 ITEM-NUMBER
   1    24 ITEM-DESCRIPTION
   1     6 COST                              999.99
```

```
PAGE-FOOTING
  NEWLINES(2)
         2 BATCH-NO
    48   5 ORDER-NUMBER
BATCH-FOOTING
  NEWLINES(2)
        70 ALL "-"
  NEWLINES(2)
     2     "BATCH NO."
         4 BATCH-NO
     8     "ENTERED  CALCULATED  ITEM-NAME"
  NEWLINES(2)
    20     "DC   "
         5 DOCUMENT-COUNT                  ENTERED
     5   5 DOCUMENT-COUNT                  CALCULATED
     4     "DOCUMENT-COUNT"
  NEWLINE
    20     "CT   "
         5 QUANTITY-ORDERED                ENTERED
     5   5 QUANTITY-ORDERED                CALCULATED
     4     "QUANTITY-ORDERED"
  NEWLINE
    20     "CT   "
         5 COST                            ENTERED
     5   5 COST                            CALCULATED
     4     "COST"
  NEWLINE
    20     "HT   "
         5 ITEM-NUMBER                     ENTERED
     5   5 ITEM-NUMBER                     CALCULATED
     4     "ITEM-NUMBER"
  NEWLINES(2)
        70 ALL "*"
  NEWLINE
```

Note. The format used above is as follows:

either

- No. Spaces before item (Ø, 0 are equivalent)
- Size of item (Ø, 0 are equivalent)
- Name of item (possibly qualified)
- PICTURE to be used (optional)

or

- No. Spaces before item
- Size of item
- Value of item (in quotes)

The terms: 'ENTERED' and 'CALCULATED' are only meaningful for a BATCH-FOOTING description.

or

- NEWPAGE (skip to a new page)
- NEWLINE (skip to a new line)
- NEWLINE(n) (skip down n lines)

CONTROL-HEADINGs and CONTROL-FOOTINGs may also be defined in the format described above. The user is left to code control-break detection.

(code produced)

(i) WORKING-STORAGE SECTION

Report Heading

```
  01    EPG-RH1.
        02   FILLER    PICTURE X(020) VALUE SPACES.
        02   FILLER    PICTURE X(039) VALUE "S T O C K   C O N T
-           "R O L   S Y S T E M".

  01    EPG-RH2.
        02   FILLER    PICTURE X(020) VALUE SPACES.
        02   FILLER    PICTURE X(039) VALUE "= = = = =   = = = =
-           "= = =   = = = = = =".

  01    EPG-RH3.
        02   FILLER    PICTURE X(027) VALUE SPACES.
        02   FILLER    PICTURE X(023) VALUE "EDITOR FOR SALES ORD
-           "ERS".

  01    EPG-RH4.
        02   FILLER    PICTURE X(028) VALUE SPACES.
        02   R-DATE    PICTURE X(008).
        02   FILLER    PICTURE X(003) VALUE SPACES.
        02   R-TIME    PICTURE X(008).
```

Similar record-descriptions will be produced for the other report lines.

(ii) PROCEDURE DIVISION

The reporting routines will be generated to produce the headings, detail, and footings.

Sample report

```
                    S T O C K   C O N T R O L   S Y S T E M
                    = = = = =   = = = = = = =   = = = = = =

                          EDITOR FOR SALES ORDERS

                             6/7/78    9:55 AM

**************************************************************************

  BATCH NO.   1      DEPT FROM DISPATCH        NUMBERED FROM  F9579
  DATE  17/6/78           TO   AUDIT                     TO    F9583

  DOCUMENTS   5      CONTROL-TOTALS            HASH-TOTALS
                       294 QUANTITY-ORDERED     1580 ITEM-NUMBER
                       104 COST

C O-DATE O-NO. QNTY I-NO. ITEM-DESCRIPTION          COST

6 020575 F9579   22   200 GREEN MANUKA BEETLE        0.22
6 020575 F9580  121   240 JESSIE NO. 2               0.22
6 020575 F9581   46   460 SILVER DOCTOR              0.22
6 020575 F9582   10   540 WICKHAMS FANCY             0.22
6 020575 F9583   95   140 PARSONS GLORY - GREEN      0.16

--------------------------------------------------------------------------

  BATCH NO.   1          ENTERED  CALCULATED  ITEM-NAME

  DOCUMENTS   5     CT      294        294     QUANTITY-ORDERED
                    CT      104        104     COST
                    HT     1580       1580     ITEM-NUMBER

**************************************************************************
```

Note. The reporting functions could be implemented using COBOL Report Writer but this is unadvisable as many COBOL compilers do not support this language extension.

- User working storage

(default code)

none

(input specification)

```
EPG-USER-WORKING-STORAGE
   any user data definitions
   up to the next EPG heading
```

(code produced)

user data definitions

Note. The user data definitions will be included in the WORKING-STORAGE SECTION without any checking by the program generator.

A later implementation may allow the same general record-description formats to be used in this section as has been defined for other DATA DIVISION specifications.

(4) PROCEDURE DIVISION

(default code)

- Skeleton code to control the working of the editor
- Code for the specifications described earlier. (See section 7.5 structure of the Generated Editor, for more detail on PROCEDURE DIVISION code).

(input specification)

EPG-USER-PROCEDURES

any user-written procedures or sections.

(code produced)

user-written procedures or sections.

Note. At any stage throughout the input specifications it is possible for the user to insert code. The method used will be to bracket the user-written code by EPG-BEGIN-USER-CODE and EPG-END-USER-CODE (in brief EPG-BEGIN and EPG-END). The code within these two headings will be inserted in the generated program as is. The other headings EPG-division or section name can be used to correctly locate this extra code.

7.5 Structure of the Generated Editor

Implementation recommendations for the overall structure of the generated editor are discussed in this section. Included are descriptions of the ordering of routines, the control/hierarchy of checking levels, and the method of assembling the generated editor.

7.5.1 The ordering of routines

The modularity of the editor can be greatly improved by the logical organization of the program structure. Section 7.4.1 outlined various input specifications with sections within the four COBOL DIVISIONS. Other modular program segments were described in more detail in section 7.4.3. Important for improved modularity is the ordering of routines within the PROCEDURE DIVISION. Listed below in table 7.6 are some suggested logical program divisions for the editor. Any working variables required by a module should be declared together as a unit within the WORKING-STORAGE SECTION, and their use should ideally be restricted to that module. The PROCEDURE DIVISION code should also be organized as units, i e in SECTIONS or groups of adjacent SECTIONS.

Table 7.6 List of logical modules for the editor

Module	Example
Control procedures	Procedures for batch control
Generalized check routines	Date check
File I/O	I/O of TRANS-REJECTED file
Error Control	Selecting error messages
Reporting	Printing transaction detail with page & control breaks
Working routines	Initialization of tables
User procedures	Alternative checks

7.5.2 Control procedures module

This module should contain the main control procedures formalizing the generalized edit algorithm, i e to input transactions, to validate them, to transfer them to auxiliary storage, and to output results as a report. A top-down structural approach should be used to control the hierarchy of checking levels. Each checking level should be controlled by a sub-module which is only generated if the particular level of check is required. (Some modular techniques were discussed in section 6.3). All other procedure modules are evoked by the control module.

The structure diagrams included as Fig 7.7 to 7.9, show a general solution to the control module.

7.5.3 Generalized check routines module

Those generalized checking routines not included as specific code within the control procedures can be collected together in this module. The 'date check' routines are a good example. If they were included in the source-code wherever a date has to be validated then a large program could result. By including them once only to be used for all date editing this possible proliferation of code is minimized.

Routines located in this module are usually parameterized. The input and output parameter variables are used to communicate with the calling routine. Input parameters are initialized by the calling routine before the generalized routine is evoked. Output parameters are set within the generalized routines before returning to the calling routine. For example, in date editing, the date to be validated is moved to a WORKING-STORAGE variable as an input parameter and the result of the check is returned in a separate variable as an output parameter. The

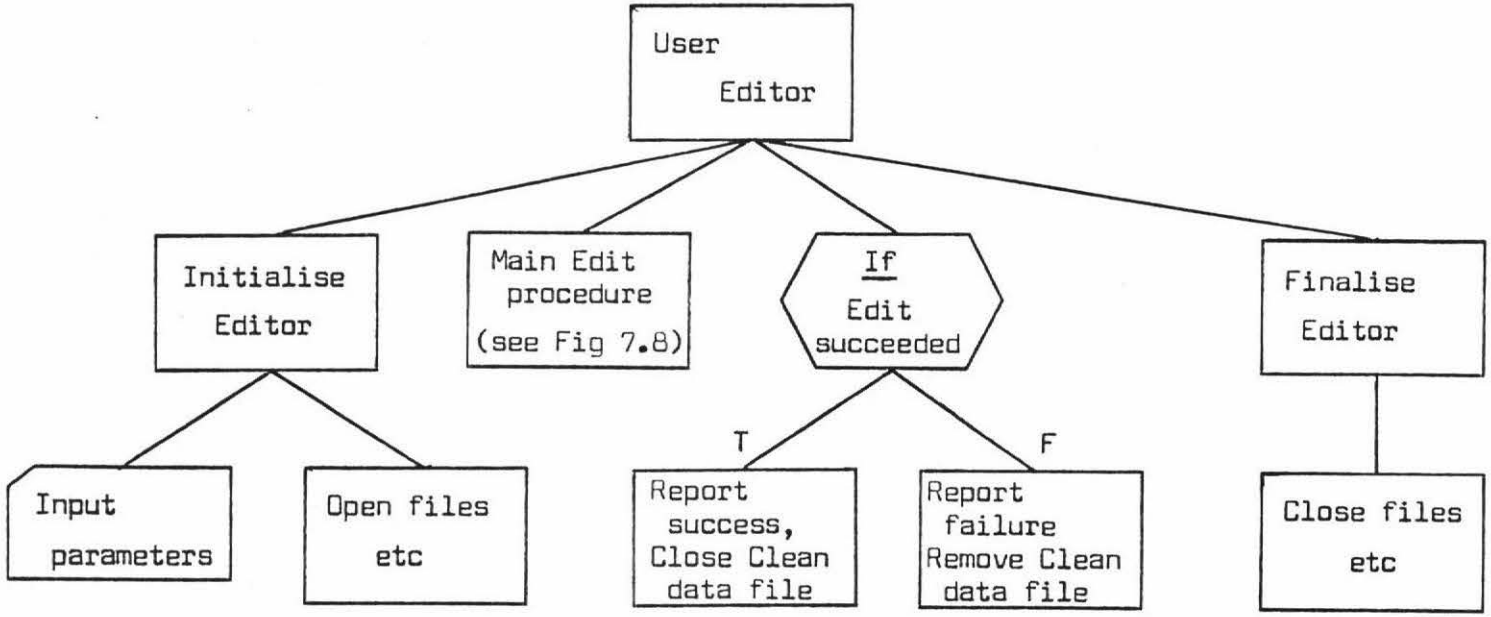


Fig 7.7 Structure of the control module for the generalized editor.

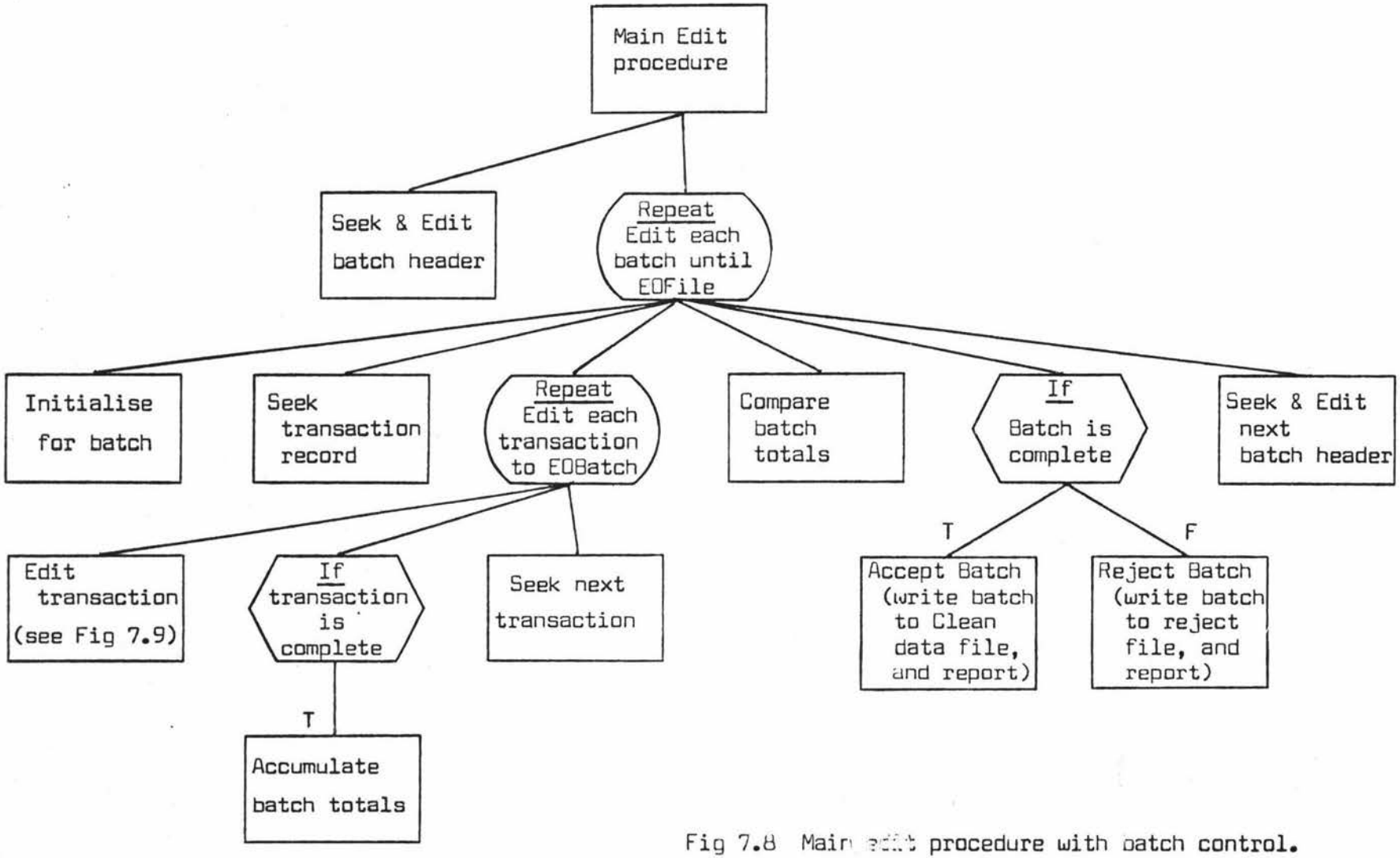


Fig 7.8 Main edit procedure with batch control.

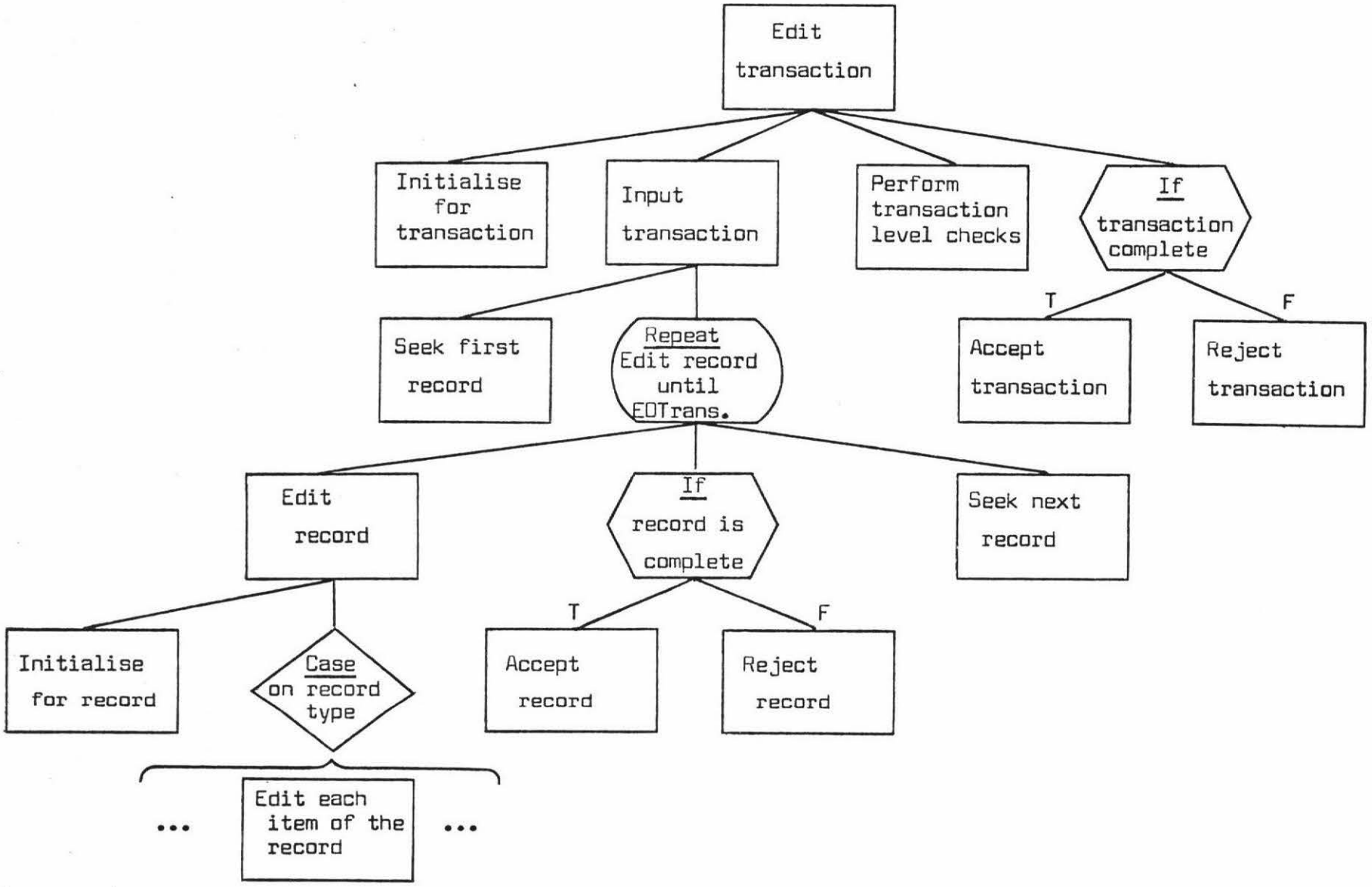


Fig 7.9 Edit procedure for each transaction.

overhead of moving data into and out of parameters must be weighed against the overhead of including specific code wherever it is required. As a general rule, if a checking function requires more than five or six simple statements or more than one paragraph of statements then it should be generalized as a parameter driven routine.

When the editor is being generated only those generalized routines that are required, should be included, i e if no 'date checking' is specified then there is no need for the date editing routines to be copied into this module of the generated program.

7.5.4 File Input/Output module

Routines to handle input/output requests should be contained in this module. Requests such as: read the next record, or write the next record are examples. Although these operations are simple in concept the routines to perform them may need to cater for complex conditions. For example, different source-code is required for different I/O device types. Sequential access and random access file organisations may also require different source-code. A future development of the software could include interactive I/O routines for on-line editing.

The files for which these routines should be provided include the unedited transactions input, the rejected transactions input, the rejected transactions output, and the clean transactions output. Routines for the edit report output file can be isolated as a separate module. (See section 7.5.6).

7.5.5 Error control module

The control of error message reporting and the course of further execution can be largely restricted to this module.

An input parameter should be used to select the appropriate error message. If all the error messages are kept in a WORKING-STORAGE table then the user could change any of the text if desired. Identifying the item in error by name or location should accompany the error message.

After an error has been reported this module should set an output parameter to indicate the severity of the error. Different actions will then be taken by the control module. Some errors will necessitate terminating the program immediately, others will not alter the course of further execution at all. The latter could arise if the user only requires the reporting of a warning. In between these extremes a variety of rejection action may need to be specified by the error module. For example, specifying that the transaction being edited should be rejected with no further item validation, or specifying that the transaction being edited should be rejected but the remaining items should be validated beforehand. Similarly specifying batch rejection could also be necessary.

7.5.6 Reporting module

This module should contain all the routines to handle output requests for the edit report file. The routines would be similar to those in the file I/O module discussed in section 7.5.4. They will however need to be substantially more complex. Headings, footings and control breaks need to be generated as well as the transferring of transaction detail to be printed. These routines should function in the same

manner as the COBOL Report Writer. The report module will also need to print error messages.

7.5.7 User procedures module

Specialized code supplied by the user should be located in this module. The program generator should insert the calls to these routines in the Control module. If the user routine performs an editing task then a success/failure parameter would also be necessary. For simplicity the same parameter could be used with all user procedures, e g called EPG-UP-RESULT with a signed numeric picture of S99. It would be the user's responsibility to ensure that this flag is set to a success or failure value before returning to the Control module. The severity of an error detected by the user procedure could also be included with the return value. Listed below is a feasible convention for communicating the user procedure result. This convention could also be used for the standard check routines produced by the generator.

0 - Success
1 - Failure (warning only)
2 - Failure (reject transaction after validating all of it)
3 - Failure (reject transaction, discontinue validating it)
4 - Failure (reject batch after validating all of it)
5 - Failure (reject batch, discontinue validating it)
6 - Failure (discontinue the edit)

A useful technique for checking that the user has obeyed this convention would be to initialize it to -1 before the call and to check it has a value between 0 and 6 inclusive after the call. Interrogating the type of failure result could be left to the Error

Control module.

7.5.8 Working routines module

This module is intended to contain all extra routines not otherwise located. For example, the routines shared by two or more modules would be collected into this section of the editor. Routines for keeping edit statistics are another example.

7.6 Assembling the Modules

Each of the modules described in this chapter will be defined simultaneously, i e as each user specification is parsed, code for several sections of the editor will be produced. As a result, the program cannot be generated sequentially.

Multiple passes of the input specifications could be avoided if multiple files are used for building up the program. After the input specifications have been scanned each file will be read and written to one file. This file will be the final product of the generalized edit program generator.

Alternatively the identification field (columns 73 to 80) could be numbered before writing each generated line to a temporary file. The first two digits of this number could be a section number and the remaining digits a section running sequence number. Sorting the temporary file will correctly order the generated editor.

REFERENCES

1. Burroughs,
B6700/B7700 DMSII, Data & Structure Definition Language (DASDL) Reference Manual (Form - 5001084).

2. SPL,
Systems and Programs (NZ) Ltd, PROGENI User Reference Manual.

3. Jancura, E
Audit and Control of Computer Systems, Mason/Charter Publishers, pg 47, 50, 51, 1974.

4. Burroughs,
AD 412, Audit Entry Data Preparation System.

5. Philippakis, A
A Popularity Contest for Languages, Datamation, pg 81, 86, 87, December 1977.

6. Kindred, A
Data Systems and Management - An Introduction to Systems Analysis and Design, pg 250, 251, Prentice-Hall, 1973.

7. Derrick, P
Data Dictionary, British Rail Technical Design Branch, (ref 7566763).

BIBLIOGRAPHY

Burroughs

B7000/B6000 Series COBOL Reference Manual, form 5001464, 1977.

Burroughs

B1700 Systems COBOL Reference Manual, form 1057197, 1976.

Chapin, N

Computers A Systems Approach, Van Nostrand Reinhold, 1971.

Davis, G

Computer Data Processing, Second Edition, McGraw-Hill, 1969.

Feingold, C

Fundamentals of COBOL Programming, Second Edition, Wm C Brown Company Publishers, 1974.

IBM

1130 COBOL Language Specifications Manual, Second Edition, form SH20-0816-1, 1971.

Jancuar, E

Audit and Control of Computer Systems, Mason/Charter Publishers, 1974.

Keppel, P

Security Consultant, Chubb Lock & Safe Co Ltd, Physical Security, Data Security, Software Protection. pg 56-62.

Kindred, A

Data Systems and Management - An Introduction to Systems Analysis and Design, Prentice-Hall, 1973.

London, K

Decision Tables, Auerbach, 1972.

Lott, R

Basic Data Processing, Prentice-Hall, 1967.

Meadow, C

Applied Data Management, John Wiley & Sons, 1976.

Neilson, M

A C P Payroll Package Input Edit, form CP0301, 1974.

Parkin, A

COBOL For Students, Edward Arnold, 1975.

Philippakis, A and Kazmier, L

COBOL for Business Applications, McGraw-Hill, 1973.

Scoular, T

Partner, Barr, Burgess & Stewart, Chartered Accountants, Audit Considerations, pg 86-94.

Watson, S

Manager, Computer Bureau (Wellington) Ltd, Protection Against Malfunction & Programming Errors & Consequential Loss, Data Privacy, pg 64, 65.

Whalen, C

RPG I and RPG II, Addison-Wesley, 1974.

APPENDIX A ANSI COBOL RESERVED WORDS

ACCEPT
ACCESS
ADD
ADDRESS
ADVANCING
AFTER
ALL
ALPHABETIC
ALTER
ALTERNATE
AND
ARE
AREA
AREAS
ASCENDING
ASSIGN
AT
AUTHOR

BEFORE
BLANK
BLOCK
BY

CF
CH
CHARACTERS
CLOCK-UNITS
CLOSE
COBOL
CODE
COLUMN
COMMA
COMP
COMPUTATIONAL
COMPUTE
CONFIGURATION
CONTAINS
CONTROL
CONTROLS
COPY
CORR
CORRESPONDING
CURRENCY

DATA
DATE-COMPILED
DATE-WRITTEN
DE
DECIMAL-POINT
DECLARATIVES
DEPENDING
DESCENDING
DETAIL
DISPLAY
DIVIDE
DIVISION
DOWN

ELSE
END
ENTER
ENVIRONMENT
EQUAL
ERROR
EVERY
EXIT

FD
FILE
FILE-CONTROL
FILLER
FINAL
FIRST
FOOTING
FOR
FROM

GENERATE
GIVING
GO
GREATER
GROUP

HEADING
HIGH-VALUE
HIGH-VALUES

I-O
I-O-CONTROL
IDENTIFICATION
IF
IN IN
INDEX
INDEXED
INDICATE
INITIATE
INPUT
INPUT-OUTPUT
INSTALLATION
INTO
INVALID
IS

JUST
JUSTIFIED

KEY

LABEL
LAST
LEADING
LEFT
LESS
LIMIT
LIMITS
LINE
LINE-COUNTER
LINES
LOCK
LOW-VALUE
LOW-VALUES

MEMORY
MODE
MODULES
MOVE
MULTIPLE
MULTIPLY

NEGATIVE
NEXT
NO
NOT
NUMBER
NUMERIC

OBJECT-COMPUTER
OCCURS
OF
OFF
OMITTED
ON
OPEN
OPTIONAL
OR OUTPUT

PAGE
PAGE-COUNTER
PERFORM
PF
PH
PIC
PICTURE
PLUS
POSITION
POSITIVE
PROCEDURE
PROCEED
PROGRAM-ID

QUOTE
QUOTES

RANDOM
RD
READ
RECORD
RECORDS
REDEFINES
REEL
RELEASE
RENAMES
REPLACING
REPORT
REPORTING
REPORTS
RERUN
RESERVE
RESET
RETURN
REVERSED
REWIND
RF
RH
RIGHT
ROUNDED
RUN

SAME
SD
SEARCH
SECTION
SECURITY
SEGMENT-LIMIT
SELECT
SENTENCE
SEQUENTIAL
SET
SIGN
SIZE
SORT
SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
STATUS
STOP
SUBTRACT
SUM
SYNC
SYNCHRONIZED

TALLYING
TAPE
TERMINATE

THAN
THROUGH
THRU
TIMES
TO
TYPE

UNIT
UNTIL
UP
UPON
USAGE
USE
USING

VALUE
VALUES
VARYING

WHEN
WITH
WORDS
WORKING-STORAGE
WRITE

ZERO
ZEROES
ZEROS

APPENDIX B ANSI COBOL 68-74 SUBSET LANGUAGE FORMAT

Conventions:

1. Words presented in uppercase are always reserved COBOL words.

2. Uppercase words which are underlined are words that are required in the type of program statement being described. Uppercase words that are not underlined are optional and are used only to improve the readability of the program.

3. Lowercase words are used to indicate the points at which data-names or constants are to be supplied by the programmer. In addition to the words 'data-name' and 'literal' the term 'identifier' is used to indicate a data-name. Uniqueness of data-names can be established through qualification.

   Other programmer supplied symbols, printed in lowercase include:

   file-name
   record-name
   integer
   formula
   condition
   statement
   any imperative statement
   any sentence

4. Items enclosed in braces {} indicate that one of the enclosed items must be used.

5. Items enclosed in brackets [] indicate that the items are optional, and one of them may be used, at the option of the programmer.

6. Three periods ... indicate that the preceding language option may be repeated several times by the user.

Identification division

```
IDENTIFICATION DIVISION.
PROGRAM-ID. program-name.
[AUTHOR. [comment-entry] . . . ]
[INSTALLATION. [comment-entry] . . . ]
[DATE-WRITTEN. [comment-entry] . . . ]
[DATE-COMPILED. [comment-entry] . . . ]
[SECURITY. [comment-entry] . . . ]
```

Environment division

```
ENVIRONMENT DIVISION.
```

Configuration section

```
CONFIGURATION SECTION.
```

Source-computer

Format 1:

```
SOURCE-COMPUTER. COPY library-name
    [ REPLACING word-1 BY { word-2      }
                          { identifier-1}
                          { literal-1   }
      [ word-3 BY { word-4      } ] . . . ].
                  { identifier-2}
                  { literal-2   }
```

Format 2:

```
SOURCE-COMPUTER. computer-name.
```

Object-computer

Format 1:

```
OBJECT-COMPUTER. COPY library-name
    [ REPLACING word-1 BY { word-2      }
                          { identifier-1}
                          { literal-1   }
      [ word-3 BY { word-4      } ] . . . ].
                  { identifier-2}
                  { literal-2   }
```

Format 2:

```
OBJECT-COMPUTER. computer-name
    [ MEMORY SIZE integer { WORDS      } ]
                          { CHARACTERS }
                          { MODULES    }
  [ SEGMENT-LIMIT IS priority-number]
```

Special-names

Format 1:

```
SPECIAL-NAMES. COPY library-name
    [ REPLACING word-1 BY { word-2      }
                          { identifier-1 }
                          { literal-1    }
      [ word-3 BY { word-4      } ] ... ].
                  { identifier-2 }
                  { literal-2    }
```

Format 2:

```
SPECIAL-NAMES. [implementor-name
    { IS mnemonic-name [ ON STATUS IS condition-name-1
    { IS mnemonic-name [ OFF STATUS IS condition-name-2
    { ON STATUS IS condition-name-1
    { OFF STATUS IS condition-name-2
    | OFF STATUS IS condition-name-2]]  }
    | ON STATUS IS condition-name-1]]   } ] ...
    | OFF STATUS IS condition-name-2]   }
    | ON STATUS IS condition-name-1]    }
    | CURRENCY SIGN IS literal] [ DECIMAL-POINT IS COMMA].
```

Input-output section

```
INPUT-OUTPUT SECTION.
```

File-control

Format 1:

```
FILE-CONTROL. COPY library-name
    [ REPLACING word-1 BY { word-2      }
                          { identifier-1 }
                          { literal-1    }
      [ word-3 BY { word-4      } ] ... ].
                  { identifier-2 }
                  { literal-2    }
```

Format 2:

```
FILE-CONTROL. {SELECT [OPTIONAL] file-name
  ASSIGN TO [integer-1] implementor-name-1 [ implementor-name-2] ...
    [ RESERVE { integer-2 } [ AREA  ] ]
              { NO        } [ AREAS ]
    [ ACCESS MODE IS { SEQUENTIAL } ]
                     { RANDOM     }
        [ RECORD KEY IS data-name ].
```

I-O-Control

Format 1:

```
I-O-CONTROL. COPY library-name
    [ REPLACING word-1 BY { word-2      }
                          { identifier-1 }
                          { literal-1    }
      [ word-3 BY { word-4      } ] ... ].
                  { identifier-2 }
                  { literal-2    }
```

Format 2:

```
I-O-CONTROL. [ RERUN [ ON {file-name-1
                           implementor-name} ]
      EVERY { {[END OF] {REEL
                         UNIT}     } OF file-name-2 } ] ...
              {integer-1  RECORDS  }
              integer-2 CLOCK-UNITS
              condition-name

   [ SAME [{RECORD
            SORT}] AREA FOR file-name-3 {, file-name-4} ... ] ...

   [ MULTIPLE FILE TAPE CONTAINS file-name-5 [POSITION integer-3]
   [ file-name-6 [POSITION integer-4] ] ... ] ....
```

Data division

```
    DATA DIVISION.
```

File section

```
    FILE SECTION.
```

File description

Format 1:

```
   FD file-name  COPY library-name
      [ REPLACING word-1 BY { word-2
                              identifier-1
                              literal-1 }
      [ word-3 BY { word-4
                    identifier-2
                    literal-2 } ] ... ] .
```

Format 2:

```
   FD file-name

      [ BLOCK CONTAINS [integer-1 TO] integer-2 { RECORDS
                                                  CHARACTERS } ]
      [ DATA { RECORD IS
               RECORDS ARE } data-name-1 [ data-name-2] ... ]
        LABEL { RECORD IS
                RECORDS ARE } { STANDARD
                                OMITTED }
      [ RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS]
      [ VALUE OF data-name-5 IS { data-name-6
                                  literal-1 }
      [ data-name-7 IS { data-name-8
                         literal-2 } ] ... ] .
      [ { REPORT IS
          REPORTS ARE } report-name-1 [ report-name-2] ... ]


      record descriptions .
```

Sort description

Format 1:

```
SD file-name COPY library-name
    [ REPLACING word-1 BY { word-4      }
                          { identifier-1 }
                          { literal-1    }
      [ word-3 BY { word-4      } ] ... ] .
                  { identifier-2 }
                  { literal-2    }
```

Format 2:

```
SD file-name
    [ DATA { RECORD IS  } data-name-1 [ data-name-2] ... ]
           { RECORDS ARE }
  [ RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS]
```

Working storage section

```
WORKING-STORAGE SECTION.

record descriptions
```

Report section

```
REPORT SECTION.
```

Report description

Format 1:

```
RD report-name COPY library-name
    [ REPLACING word-1 BY { word-2      }
                          { identifier-1 }
                          { literal-1    }
      [ word-3 BY { word-4      } ] ... ] .
                  { identifier-2 }
                  { literal-2    }
```

Format 2:

```
RD report-name
  [ CODE mnemonic-name-1]
  [ { CONTROL IS  } { FINAL                                    } ]
    { CONTROLS ARE } { identifier-1 [ identifier-2] ...        }
                     { FINAL identifier-1 [ identifier-2] ...  }
  [ PAGE [ LIMIT IS  ] integer-1 { LINE  } [ HEADING integer-2]
         [ LIMITS ARE ]          { LINES }
  [ FIRST DETAIL integer-3] [ LAST DETAIL integer-4]
  [ FOOTING integer-5] ].
```

Record description

Format 1:

```
01 data-name-1  COPY library-name
    [ REPLACING word-1 BY { word-2       } ]
                          { identifier-1 }
                          { literal-1    }
    [   word-3 BY { word-4       } ] ... ] .
                  { identifier-2 }
                  { literal-2    }
```

Format 2:

```
level-number { data-name-1 } [ REDEFINES data-name-2]
             { FILLER      }

    [ { PICTURE } IS character-string ] [ [ USAGE IS] { COMPUTATIONAL } ]
      { PIC     }                                     { COMP          }
                                                      { DISPLAY       }
                                                      { INDEX         }

    [ OCCURS { integer-1 TO integer-2 TIMES [DEPENDING ON data-name-3] }
             { integer-2 TIMES                                         }
    [ { ASCENDING  } KEY IS data-name-4 [ data-name-5] ... ] ...
    | { DESCENDING }
      INDEXED BY index-name-1 [ index-name-2] ... ] ]

    [ { SYNCHRONIZED } [ LEFT  ] ] [ { JUSTIFIED } RIGHT ]
      { SYNC         } [ RIGHT ]   { JUST      }

                                                  [ BLANK WHEN ZERO]

    [ VALUE IS literal-3].
```

Format 3:

```
66 data-name-1  RENAMES data-name-2 [THRU data-name-3].
```

Format 4:

```
88 condition-name
    { VALUE IS  } literal-1 [THRU literal-2]
    { VALUES ARE }
  [ literal-3 [THRU literal-4] ] ... .
```

Report group description

Format 1:

```
01 data-name-1  COPY library-name

  [                              { word-2       } ]
  [ REPLACING word-1 BY          { identifier-1 } ]
  [                              { literal-1    } ]

  [             { word-4       } ]     ]
  [ word-3 BY   { identifier-2 } ] ... ] .
  [             { literal-2    } ]     ]
```

Format 2:

```
01 [data-name-1]

  [                        { integer-1       } ]
  [ LINE NUMBER IS         { PLUS integer-2  } ]
  [                        { NEXT PAGE       } ]

  [                        { integer-3       } ]
  [ NEXT GROUP IS          { PLUS integer-4  } ]
  [                        { NEXT PAGE       } ]

               { REPORT HEADING                                      }
               { RH                                                  }
               { PAGE HEADING                                        }
               { PH                                                  }
               { { CONTROL HEADING }   { identifier-1 }              }
               { { CH              }   { FINAL        }              }
               { DETAIL                                              }
  TYPE IS      { DE                                                  }
               { { CONTROL FOOTING }   { identifier-2 }              }
               { { CF              }   { FINAL        }              }
               { PAGE FOOTING                                        }
               { PF                                                  }
               { REPORT FOOTING                                      }
               { RF                                                  }

  [ [USAGE IS] DISPLAY].
```

Format 3:

```
level-number [data-name-1]
  [  BLANK WHEN ZERO]
  [  COLUMN NUMBER IS integer-1]
  [  GROUP INDICATE]
  [  { JUSTIFIED }  RIGHT ]
  [  { JUST      }        ]

  [                        { integer-2       } ]
  [  LINE NUMBER IS        { PLUS integer-3  } ]
  [                        { NEXT PAGE       } ]

  [  { PICTURE }  IS character-string ]
  [  { PIC     }                      ]

  [  RESET ON  { identifier-1 } ]
  [            { FINAL        } ]

  {  SOURCE IS identifier-2                                        }
  {  SUM identifier-3 [ identifier-4] . . . [UPON data-name-2]     }
  {  VALUE IS literal-1                                            }
  [  [USAGE IS] DISPLAY].
```

Procedure division

```
PROCEDURE DIVISION.

[sections  ]
[paragraphs]
```

Declaratives

```
DECLARATIVES.
  {section-name SECTION. declarative-sentence
  {paragraph-name. {sentence} . . . } . . . } . . .
  END DECLARATIVES.
```

Section names

```
{section-name SECTION [priority-number].
{paragraph-name. {sentence} . . . } . . . } . . .
```

Paragraph names

```
{paragraph-name. {sentence} . . . } . . .
```

Accept

```
ACCEPT identifier [FROM mnemonic-name]
```

Add

Format 1:

```
ADD {identifier-1} [ identifier-2 ]
    {literal-1   } [ literal-2    ] . . . TO identifier-m [ROUNDED]
  [ identifier-n [ROUNDED]] . . .
  [ ON SIZE ERROR imperative-statement]
```

Format 2:

```
ADD {identifier-1} {identifier-2} [ identifier-3 ] . . .
    {literal-1   } {literal-2   } [ literal-3    ]
  GIVING identifier-m [ROUNDED]
  [ ON SIZE ERROR imperative-statement]
```

Format 3:

```
ADD {CORRESPONDING} identifier-1 TO identifier-2 [ROUNDED]
    {CORR          }
  [ ON SIZE ERROR imperative-statement]
```

Alter

```
ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2
[ procedure-name-3 TO [PROCEED TO] procedure-name-4] . . .
```

Close

```
CLOSE file-name-1 [REEL  ] [WITH {NO REWIND}]
                  [UNIT  ]      {LOCK     }
  [ file-name-2 [REEL  ] [WITH {NO REWIND}]
                [UNIT  ]      {LOCK     }      . . .
```

Compute

```
COMPUTE identifier-1 [ROUNDED] = {identifier-2          }
                                 {literal-1             }
                                 {arithmetic-expression }
  [ ON SIZE ERROR imperative-statement]
```

Copy

```
COPY library-name
  [ REPLACING word-1 BY { word-2       }
                        { identifier-1 }
                        { literal-1    }
    [ word-3 BY { word-4       } ]... ]
                { identifier-2 }
                { literal-2    }
```

Display

```
DISPLAY { literal-1    } [ literal-2    ] ... [UPON mnemonic-name]
        { identifier-1 }   { identifier-2 }
```

Divide

Format 1:

```
DIVIDE { identifier-1 } INTO identifier-2 [ROUNDED]
       { literal-1    }
  [ ON SIZE ERROR imperative-statement]
```

Format 2:

```
DIVIDE { identifier-1 } INTO { identifier-2 }
       { literal-1    }     { literal-2    }
  GIVING identifier-3 [ROUNDED]
  [ ON SIZE ERROR imperative-statement]
```

Format 3:

```
DIVIDE { identifier-1 } BY { identifier-2 }
       { literal-1    }   { literal-2    }
  GIVING identifier-3 [ROUNDED]
  [ ON SIZE ERROR imperative-statement]
```

Format 4:

```
DIVIDE { identifier-1 } INTO { identifier-2 }
       { literal-1    }     { literal-2    }
    GIVING identifier-3 [ROUNDED] REMAINDER identifier-4
  [ ON SIZE ERROR imperative-statement]
```

Format 5:

```
DIVIDE { identifier-1 } BY { identifier-2 }
       { literal-1    }   { literal-2    }
  GIVING identifier-3 [ROUNDED] REMAINDER identifier-4
  [ ON SIZE ERROR imperative-statement]
```

Enter

```
ENTER language-name [routine-name].
```

Exit

```
EXIT.
```

Generate

```
GENERATE identifier
```

Go

Format 1:

```
GO TO [procedure-name-1]
```

Format 2:

```
GO TO procedure-name-1 [ procedure-name-2] ...
   procedure-name-n DEPENDING ON identifier
```

If

```
IF condition { statement-1     } [ ELSE { statement-2     } ]
             { NEXT SENTENCE }          { NEXT SENTENCE }
```

Initiate

```
INITIATE report-name-1 [ report-name-2] ...
```

Move

Format 1:

```
MOVE { identifier-1 } TO identifier-2 [ identifier-3] . . .
     { literal-1    }
```

Format 2:

```
MOVE { CORRESPONDING } identifier-1 TO identifier-2
     { CORR          }
```

Multiply

Format 1:

```
MULTIPLY { identifier-1 } BY identifier-2 [ROUNDED]
         { literal-1    }
  [ ON SIZE ERROR imperative-statement]
```

Format 2:

```
MULTIPLY { identifier-1 } BY { identifier-2 }
         { literal-1    }    { literal-2    }
  GIVING identifier-3 [ROUNDED]
  [ ON SIZE ERROR imperative-statement]
```

Open

```
                { INPUT file-name-1  { REVERSED          } }
                {                    { WITH NO REWIND    } }
OPEN            { OUTPUT file-name-3 [WITH NO REWIND]    }
                { I-O file-name-5                        }
  [   file-name-2 [ { REVERSED       } ]] . . .          ]
  [                 { WITH NO REWIND } ]]               ] . . .
  [   file-name-4 [WITH NO REWIND] ] . . .               ]
  [   file-name-6] . . .                                 ]
```

Perform

Format 1:

```
PERFORM procedure-name-1 [THRU procedure-name-2]
```

Format 2:

```
PERFORM procedure-name-1 [THRU procedure-name-2] { identifier-1 } TIMES
                                                 { integer-1    }
```

Format 3:

```
PERFORM procedure-name-1 [THRU procedure-name-2] UNTIL condition-1
```

Format 4:

```
PERFORM procedure-name-1 [THRU procedure-name-2]
  VARYING { index-name-1 } FROM { index-name-2 }
          { identifier-1 }      { literal-2    }
                                { identifier-2 }
  BY { literal-3    } UNTIL condition-1
     { identifier-3 }
  [ AFTER { index-name-4 } FROM { index-name-5 }
          { identifier-4 }      { literal-5    }
                                { identifier-5 }
  BY { literal-6    } UNTIL condition-2
     { identifier-6 }
  [ AFTER { index-name-7 } FROM { index-name-8 }
          { identifier-7 }      { literal-8    }
                                { identifier-8 }
  BY { literal-9    } UNTIL condition-3 ] ]
     { identifier-9 }
```

Read

```
READ file-name RECORD [INTO identifier]
    { AT END imperative-statement        }
    { INVALID KEY imperative-statement   }
```

Release

```
RELEASE record-name [FROM identifier]
```

Return

```
RETURN file-name RECORD [INTO identifier]
    AT END imperative-statement
```

Search

Format 1:

```
SEARCH identifier-1 [VARYING {index-name-1}]
                             {identifier-1 }
  [ AT END imperative-statement-1]
    WHEN condition-1 {imperative-statement-2}
                     {NEXT SENTENCE        }
  [ WHEN condition-2 {imperative-statement-3}]  ...
                     {NEXT SENTENCE        }
```

Format 2:

```
SEARCH ALL identifier-1 [ AT END imperative-statement-1]
    WHEN condition-1 {imperative-statement-2}
                     {NEXT SENTENCE        }
```

Set

Format 1:

```
SET {identifier-1   [ identifier-2] ... } TO {identifier-3}
    {index-name-1 [ index-name-2] ...   }    {index-name-3}
                                             {literal-1    }
```

Format 2:

```
SET index-name-1 [ index-name-2] ... {UP BY   } {identifier-1}
                                     {DOWN BY} {literal-1   }
```

Sort

```
SORT file-name-1 ON {DESCENDING} KEY data-name-1 [ data-name-2] ...
                    {ASCENDING }
  [ ON {DESCENDING} KEY data-name-3 [ data-name-4] ... ] ...
       {ASCENDING }
  {INPUT PROCEDURE IS section-name-1 [THRU section-name-2] }
  {USING file-name-2                                       }
  {OUTPUT PROCEDURE IS section-name-3 [THRU section-name-4]}
  {GIVING file-name-3                                      }
```

Stop

```
STOP {literal}
     {RUN    }
```

Subtract

Format 1:

```
SUBTRACT {literal-1   } [ literal-2   ] ...
         {identifier-1}   identifier-2 ]
  FROM identifier-m [ROUNDED] [ identifier-n [ROUNDED] ] ...
  [ ON SIZE ERROR imperative-statement]
```

Format 2:

```
SUBTRACT {literal-1   } [ literal-2   ] ...
         {identifier-1}   identifier-2 ]
  FROM {literal-m   } GIVING identifier-n [ROUNDED]
       {identifier-m}
  [ ON SIZE ERROR imperative-statement]
```

Format 3:

SUBTRACT { CORRESPONDING / CORR } identifier-1 FROM identifier-2 [ROUNDED]
[ON SIZE ERROR imperative-statement]

Terminate

TERMINATE report-name-1 [report-name-2] . . .

Use

Formate 1:

USE AFTER STANDARD ERROR PROCEDURE ON
{ file-name-1 [file-name-2] . . . / INPUT / OUTPUT / I-O }

Format 2:

USE BEFORE REPORTING identifier-1

Write

Format 1:

WRITE record-name [FROM identifier-1]
[{ BEFORE / AFTER } ADVANCING { identifier LINES / integer LINES / mnemonic-name }]

Format 2:

WRITE record-name [FROM identifier-1]
INVALID KEY imperative-statement