ICL Technical Journal

Volume 1 Issue 3 November 1979

ICL Technical Journal

Contents

Volume 1 Issue 3

Meteosat 1: Europe's first meteorological satellite
D. Ainsworth 195

An analysis of checkpointing
A. Brock 211

Statistical and related systems
B.E. Cooper 229

Structured programming techniques in interrup-driven routines
P.F. Palmer 247

The content addressable file store – CAFS
V.A.J. Maller 265

Computing in the humanities
S. Hockey 280

Erratum 291

The data dictionary system in analysis and design
T.J. Bourne 292

Notes for authors 299

ICL Technical Journal

The ICL Technical Journal is published twice a year by Peter Peregrinus Limited on behalf of International Computers Limited

All correspondence and papers to be considered for publication should be addressed to the Editor

Annual subscription rate: £5 (Details from: The Technical Journal Office, ICL House, Putney, London, SW15 1SW)

Publisher
Peter Peregrinus Limited
PO Box 8, Southgate House, Stevenage, Herts SG1 1HQ, England

Printed by A. McLay & Co. Ltd., London and Cardiff ISSN 0142–1557

Meteosat 1: Europe's first meteorological satellite

David Ainsworth
ICL Manager, European Space Agency Project, Darmstadt, W.Germany

Abstract

Meteosat is a geostationary satellite designed and equipped for the collection and transmission to Earth of meteorological and environmental data. For the control of the satellite and the processing and dissemination of this information a very large and complex system of computers and ancillary equipment has been set up in the European Space Operations Centre at Darmstadt in West Germany. Two large ICL 2980 computers form the core of the system and ICL was the main contractor for the whole installation. The paper describes the main features of the satellite and of the computing system and also of the information collected and the analyses performed.

1 Introduction

Television viewers in the British Isles and Germany are becoming accustomed to nightly pictures showing cloud over Northern Europe along with tomorrow's weather forecast. These pictures are one of the more public products of Europe's first meteorological satellite Meteosat 1.

The European Space Agency has co-ordinated European industry in the design and construction of the satellite and its associated ground facilities. The Agency is responsible for the operation of Meteosat on behalf of the European Meteorological Community. Meteosat was launched on the 23rd November 1977 into its transfer orbit off the coast of Brazil. Over the following fortnight it was moved into its operational geostationary position 36000 km above the equator and Greenwich Meridian. The first pictures were taken on the 7th December 1977 when the spacecraft was in position at 0 degrees North 0 degrees East above the coast of Africa between Gabon and Ghana.

The Meteosat system meets the main data acquisition needs of the European Meteorological services which, in turn, provide weather forecasting facilities for their user communities. The responsibility for providing the local weather forecasts, however, is the prerogative of national weather bureaux such as the British Meteorological Office at Bracknell or the German Meteorological Service at Offenbach. Meteosat also represents Europe's contribution to two programmes set up by the World Meteorological Organisation. The first is a permanent undertaking and is known as the world weather watch programme. The second is an experimental study undertaken jointly with the International Council of Scientific Unions and is known as the global atmospheric research programme (GARP).

The first GARP global weather experiment (FGGE) commenced on the 1st December 1978 and will be operational for one year. The experiment consists of a network of five geostationary satellites distributed every 72° around the equator (Fig. 1). Japan has one satellite named GMS covering Eastern Asia. The USA has two satellites GOES-W and GOES-E covering the western and eastern coasts of America. The USSR was responsible for covering Western Asia. Unfortunately, however, their satellite was not available in time for the experiment so a third GOES satellite was provided by the USA. The operational control of this satellite is provided by the European Space Agency which also collects the satellite experimental data and retransmits it to the USA for processing.

The Meteosat satellite is controlled by the Meteosat ground computer system (MGCS) which is located in the European Space Operations Centre at Darmstadt in the Federal Republic of Germany. The link is provided by a 50 km broadband landline to the Odenwald from where a 12 m diameter antenna provides the radio link to the satellite. The Meteosat system provides the following facilities for its users: earth imaging, image processing, meteorological information extraction, archiving, dissemination, and environmental data capture. In addition to these user facilities the system must also provide for the operational control of the satellite.

2 User facilities

2.1 Earth imaging

The principal payload of the satellite is a 3-channel radiometer which can provide images in three spectral bands:

(*a*) two visible channels providing images in the 0·4 - 1·1 μm spectral band
(*b*) an infrared water-vapour channel providing images in the 5·7 – 7·1 μm water-vapour-absorption band
(*c*) an infrared thermal channel providing images in the 10·5 – 12·5 μm spectral band.

The infrared image of the full Earth's disc visible from the satellite is composed of 2500 lines comprising 2500 picture elements (pixels) per line giving a spatial resolution of 5 km at the subsatellite point.

When the water vapour channel is switched off the two adjacent visible channels produce 5000 pixels per line and 5000 lines. Hence the visible spatial resolution at the subsatellite point is 2·5 km.

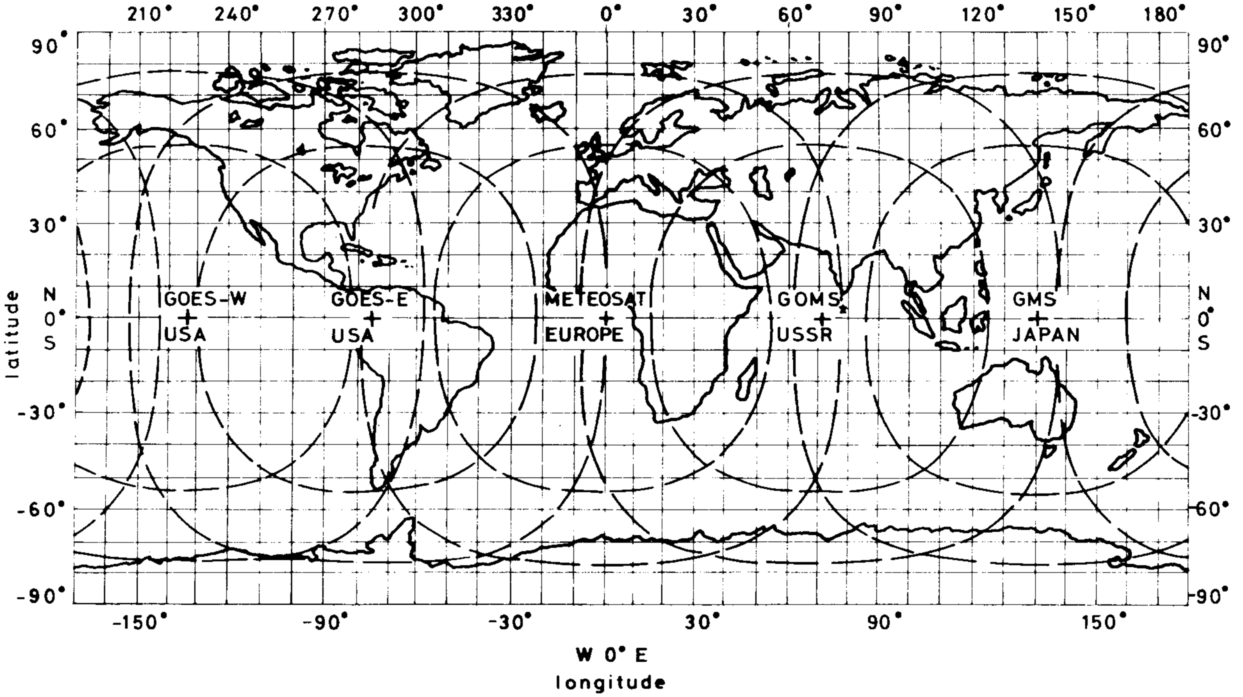


Fig. 1 Coverage of the geostationary satellites

telecommunications and image coverage
image coverage for calculation purposes

(GOMS temporarily replaced by a GOES satellite at 58^{O}E during the global weather experiment and controlled by the European Space Agency - 1st December 1978-30th November 1979)

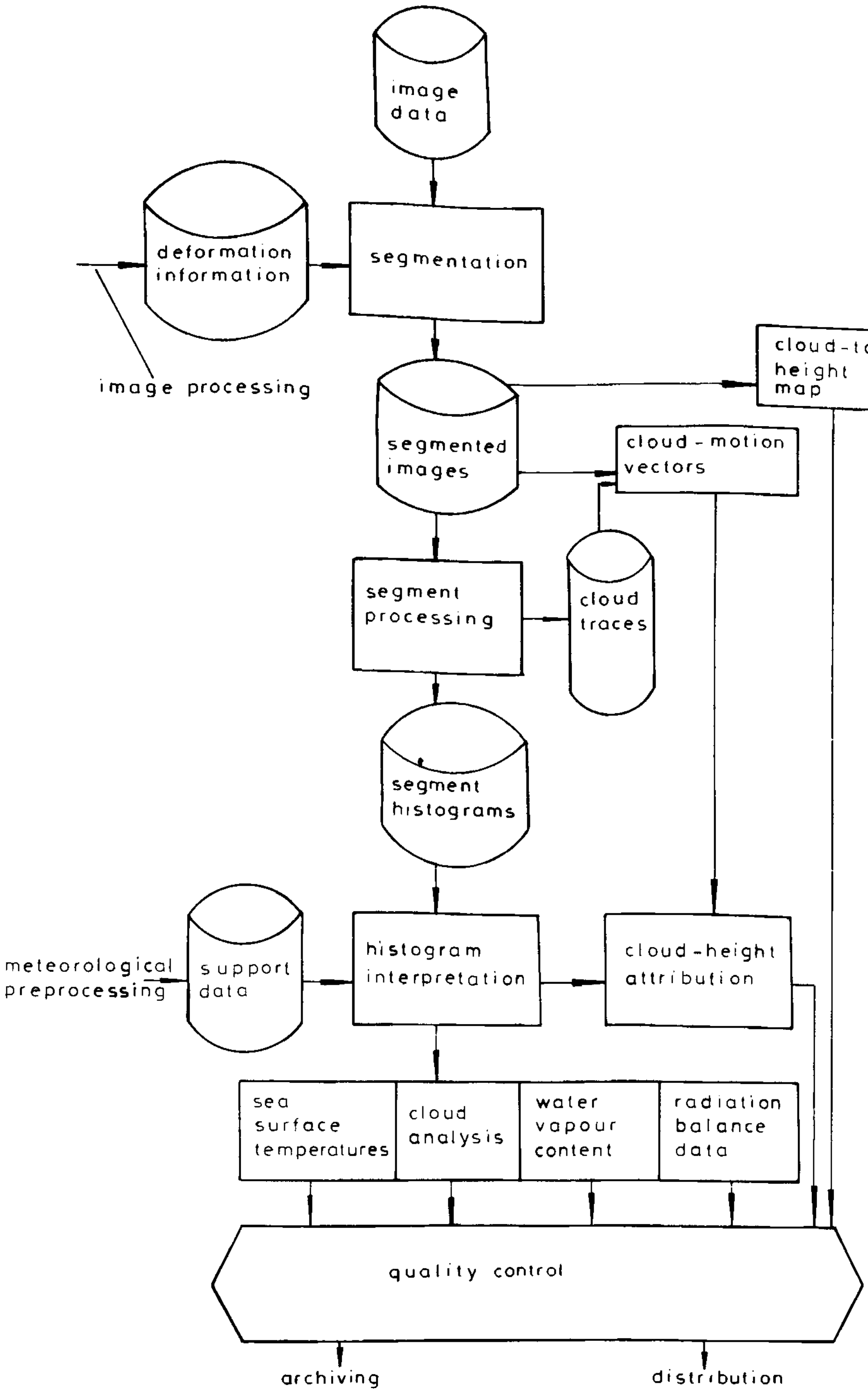


Fig. 3 MIEC processing system

within the 50° circle arc of the subsatellite point. For each of these segments a 3-stage analysis is made.

Stage 1 consists of a full description of the *a priori* status of the segment. A segment characteristics table is built up showing for example: physical characteristics such as land, sea, or sun glint; meteorological characteristics; cloud climatology; spacecraft and solar elevation and azimuth.

Stage 2 consists of a multispectral analysis of the satellite radiance data for the segment. The output is a cluster analysis in which each pixel is assigned to a specific cluster which in turn is defined in terms of its mean radiance and variance.

Stage 3 assigns each radiance cluster to a particular source of radiation such as a cloud level, land or sea. The segment can then be used as input to product processing.

(*a*) *Cloud motion vectors:* A sequence of three consecutive images is used to determine the motion of clouds from one image to the next and hence to deduce the apparent wind speeds at three altitude ranges per segment. The actual output depends upon suitable cloud traces for the method used. Individual segments from the second consecutive image are treated as target windows and a search is made over a 3 x 3 segment window in each of the adjacent images for a good match. Matching is achieved by the calculation of crosscorrelation coefficients for each possible displacement of the target window in the larger search window. The cross-correlation surfaces are analysed for significant peaks, and peak displacements are accepted as wind vectors if there is an acceptable degree of symmetry between the two displacements obtained from two image pairs.

In the calculation of wind vectors, for example, some transparent cirrus clouds present a particular problem where a high-level wind may be calculated and shown as a seemingly low-level wind at variance with other surrounding low-level winds. Partially, this can be corrected automatically using the water-vapour channels but orographic effects must still be corrected. Therefore, following automatic calculation, the wind vectors need to be validated by meteorologists at coloured video terminals.

(*b*) *Sea surface temperatures:* For each segment covering sea the 'skin' temperature of the ocean is measured. The method is based upon infrared clustered radiances converted by means of tables to temperatures. Relative temperatures can be measured to an accuracy of 1°C but correlation with other measurement sources is difficult. For example, in weather ships ocean temperature is measured by dropping a bucket over the side of the ship and putting a thermometer into the resulting contents!

(*c*) *Cloud analysis:* The cloud analysis produced represents the percentage of cloud cover in each of the segments processed.

(*d*) *Upper tropospheric humidity:* The method used is based upon the interpretation of the 6 μm water-vapour channel per segment. The results are representative of a fairly deep layer in the upper troposphere and cannot be assigned to a particular level.

(*e*) *Radiation balance data:* The production of radiation balance data is highly experimental. The objective is the evaluation of the mean radiation fluxes per segment arriving at and departing from the Earth's surface.

(*f*) *Cloud-top heights:* The objective is to provide information on the highest cloud determined in each section of 4 x 4 infrared pixels. The resulting product is a map where the height of the top is represented by one of eight grey levels. Black is used to represent sections with no cloud or no cloud tops above 3000 m. White indicates cloud tops above 12,000 m. and the remaining six grey levels correspond to layers of 1,500 m. depth between those two extremes.

2.4 Archiving

The Meteosat ground computer system provides for the archiving of all processed images in both digital and photographic formats. The data are recorded on high-density magnetic tapes at a packing density of 22,000 bit/in on each of 14 tracks. Hence all the data for one day can be stored on one tape instead of the 48 conventional magnetic tapes that would otherwise be necessary.

The photographic archive is provided on photographic negative film produced on a Vizir laser beam recorder. The film size is 425 x 460 mm and is of such a quality to reproduce 64 levels of grey scale. The laser beam has a spot diameter of 40 μm and gives some 11,000 separate points for each of 11,000 lines. Three 20 x 20 cm pictures are stored on one film thus facilitating a complete record of all images taken during a 30 min slot. In addition some full 40 x 40 cm visible pictures are archived.

2.5 Dissemination

The first five meteorological products are transmitted over the meteorological global telecommunications system (GTS). In addition however, Meteosat has two dedicated dissemination channels operating at 1·691 GHz and at 1·6945 GHz which are both used to distribute a variety of data. Two forms of transmission are used: conventional analogue transmissions (Wefax) and high-resolution digital transmission (Fig. 4).

The Wefax transmissions can be received by the simplest kind of receiving station known as a secondary data user station (SDUS). This consists essentially of a 3 m antenna, a receiver, and some form of recording device.

The digital transmission requires the more complex primary data user stations (PDUS). This consists of a 5 m antenna, receiver, frame synchroniser and a minicomputer in a basic system. To this can be added a variety of devices such as tape recorders and video terminals.

When image processing has been completed the images are cut into convenient formats which have had latitude and longitude grids and coastlines added before transmission. Digital images from all three spectral ranges can be line multiplexed in the dissemination schedule. The schedule is designed to include the entire Meteosat field of view (A) at least once every three hours and Europe (B) every half hour. For Wefax visible images the European areas (C2 and C3) are transmitted on a half-hourly basis and the rest of the field of view at intervals of 1-3 h during the daylight period. Infrared images dividing the Meteosat field of view into nine sectors are transmitted at a similar frequency but on a 24 h basis. Water-vapour images, again on a 9-sector basis, are transmitted twice daily. In total some 500 image formats are disseminated daily, including data from other geostationary spacecraft (GOES-E, and GOES-Indian Ocean).

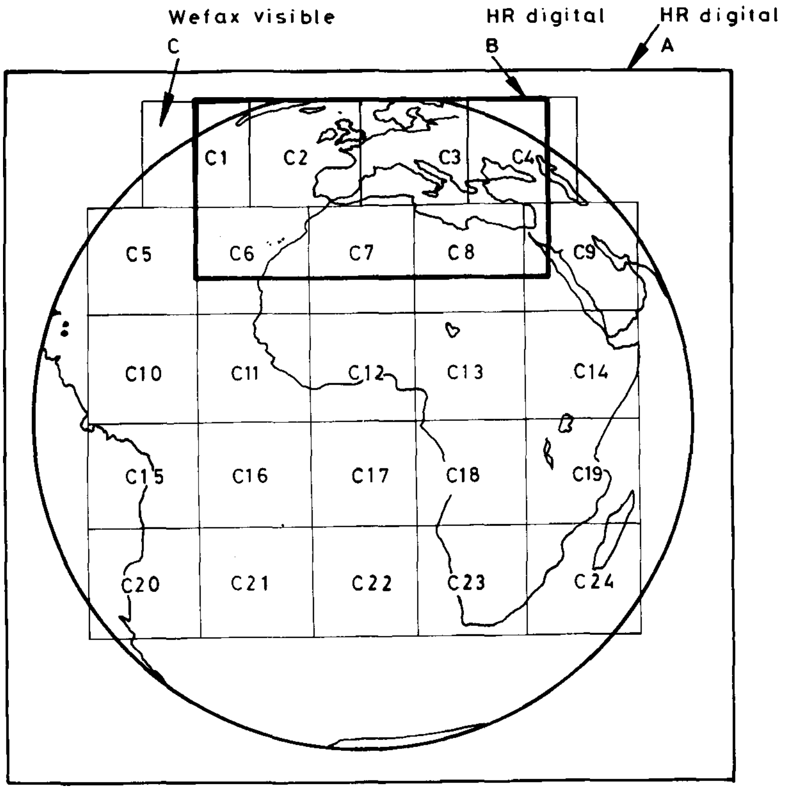


Fig. 4 Digital image formats and Wefax visible formats disseminated from ESOC

Cloud-top height maps and prognostic charts from the weather forecasting agencies will also be disseminated in Wefax format by the satellite on a scheduled basis.

In addition to the dissemination of Meteosat products high-resolution and Wefax images of the Americas are received at the Centre de Météorologie Spatiale in Lannion in France. From here they are transmitted to Meteosat and are incorporated in the dissemination schedule on a 3-hourly basis.

2.6 *Environmental data collection*

In addition to its meteorological function Meteosat also has a 66-channel facility for the collection and distribution of data collected by automatic or semiautomatic data collection platforms (DCPs). The DCP can exist in a variety of platform types such as fixed land stations, ocean buoys, ships, aircraft and balloons (Fig. 5). In this case the European Space Agency is simply a spacecraft operator with a flexible data storage and translation facility in the ICL 2980. Having had the DCP radio set certified by ESA the DCP operator is entirely responsible for the DCP operation following agreement of admission procedure. In principle each DCP is interrogated or has transmission facilities once per hour for one minute, and hence a theoretical total of over 3000 DCP connections are available.

Some current projects under consideration include:

(*a*) study of the ecological behaviour of desert plants in Namibia
(*b*) warning of ideal locust breeding conditions
(*c*) direction of fishing fleets to areas of cold water plankton in warmer seas
(*d*) unmanned hydrological stations in Greenland, the UK and Germany
(*e*) airborne weather reports using commercial airlines.

Three basic types of DCP platforms are available to cope with these studies:

(*a*) a self-timed platform which transmits at regular intervals
(*b*) an interrogated platform in response to a telecommand from MGCS
(*c*) an alert platform which monitors environmental data and transmits only when certain criteria have been exceeded.

The Agency has an active policy of encouraging the use of the DCP facility for the collection of useful environmental data.

3 **The satellite**

Now that the products available to the meteorological and scientific communities using Meteosat have been described it is interesting to consider the satellite necessary to provide these facilities (Fig. 6). Meteosat 1 is 2·1 m in diameter and 3·195 m in length. Its weight at the beginning of its life in orbit was 293 kg. This weight will gradually reduce to 245 kg as the hydrazine propellant is utilised during its lifetime. In orbit the satellite is stabilised by rotation at 100 rev/min with elaborate control of its spin axis to less than $0{\cdot}3^\circ$ of the orbit plane normal.

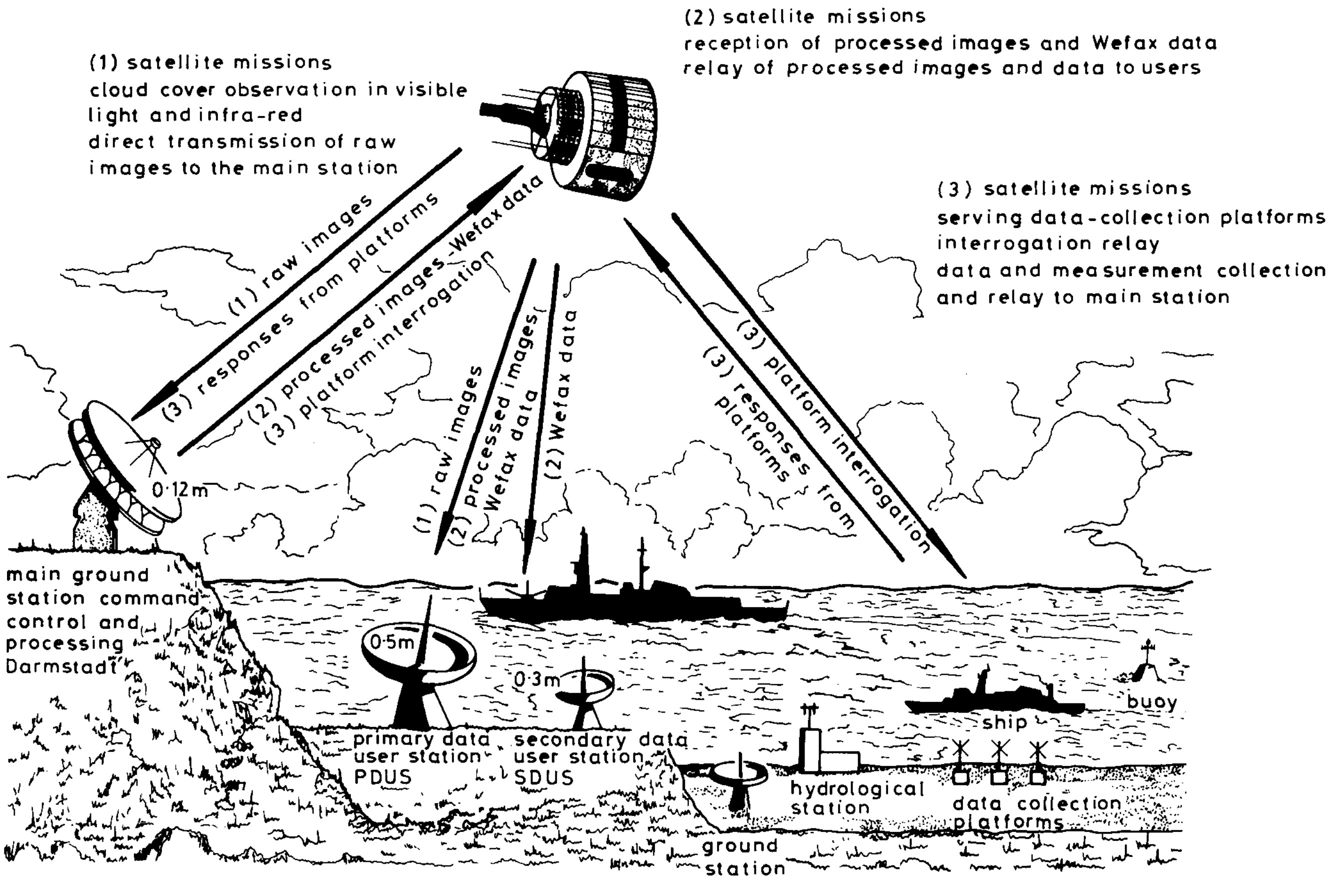

(1) satellite missions
cloud cover observation in visible light and infra-red
direct transmission of raw images to the main station
(2) satellite missions
reception of processed images and Wefax data
relay of processed images and data to users
(3) satellite missions
serving data-collection platforms
interrogation relay
data and measurement collection and relay to main station
(1) raw images
(3) responses from platforms
(2) processed images-Wefax data
(3) platform interrogation
(1) raw images
(2) processed images, Wefax data
(2) Wefax data
(3) responses from platforms
(3) platform interrogation
0·12m
main ground station command control and processing Darmstadt
0·5m
0·3m
primary data user station PDUS
secondary data user station SDUS
ground station
hydrological station
ship
buoy
data collection platforms

Fig. 5 Meteosat data flow

The spacecraft has four main thrusters together with two vernier thrusters. The system is used to control Meteosat's attitude in space and also to return Meteosat from 1° E up the geopotential slope to 1° W longitude approximately every five months. Two of the main thrusters are mounted with their thrust axes parallel to the satellite spin axes. They generate a torque about the spacecraft's centre of gravity and can be used for spin axis manoeuvres and inclination control. The other two main motors are radial thrusters acting with thrust axes through the in-orbit centre of gravity and at right angles to the spin axes. They are used for east-west station keeping and active nutation damping. The small vernier motors act at an inclination of 12° to the radial thrusters and are also offset from the centre of gravity in opposition to each other. The torques generated contribute to both spin-rate control and to active nutation damping.

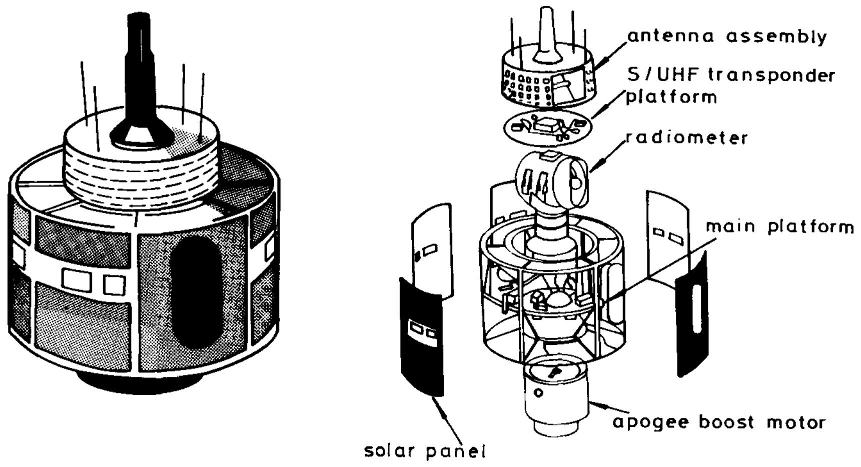


Fig. 6 Meteosat

Power for the satellite's electrical equipment is supplied by 16128 silicon solar cells mounted on six panels surrounding the main cylindrical body of the satellite. During eclipses power is supplied by a 7 Ah nickel-cadmium battery.

The Radiometer is an electro-optical device comprising a Ritchey-Chretien 40 cm aperture telescope, scanning mechanism, focusing and calibration devices and electronic detection chains. Earth images are generated at 30 min intervals. An image consists of picture elements transmitted on a line-by-line basis to the ground. An east-west line of elements is generated as the radiometer scans the Earth due to the spinning motion of the satellite. A succession of lines is obtained by racking up the radiometer telescope from south to north synchronously with the satellite spin periods in 2500 steps. As the satellite is spinning at 100 rev/min this takes 25 min. The remaining 5 min are used for resetting the mechanisms and restabilising the spacecraft.

The cylindrical body of the smaller drum-shaped body is covered with an array of radiating dipole antenna elements. The individual rows are activated in reverse sequence to the satellite spin sense so that at any point in time the five rows nearest the Earth are energised. This system constitutes an electronically despun antenna for transmission in the S-band. The data rate of the raw image signal, transmitted in stretched mode at 1686·33 MHz, is normally 166 kbit/s. During each spacecraft spin period the radiometer sees the earth from 1/20th of a resolution. The image data is then buffered in an onboard memory and transmitted to the earth during the remaining 19/20th of the resolution. Should the memory fail it can be bypassed and the raw data transmitted in burst mode at 2·7 Mbit/s. The two cylinders mounted on top of the drum are toroidal-pattern antennae for S-band and UHF, respectively. The UHF at 400-470 MHz is used to receive data from the data-collection platforms. The four thin rods forming a turnstile antenna for VHF links (137-149 MHz) were used during launch for telecommanding, telemetry, and ranging functions. During normal operations the VHF system acts as a backup to the S-band for these functions.

4 Ground control hardware

The control of the satellite is exercised by the computer system for which ICL was the prime contractor and project manager. ICL from the United Kingdom provided the main processors; CIT Alcatel from France furnished the image display subsystem; Christian Rovsing from Denmark constructed the archiving subsystem and array processors; Siemens from West Germany supplied the front-end and back-end processors; and Logica from the United Kingdom supplied the hardware interfaces (Fig. 7).

Application software was partly written directly by the Agency and partly produced by contracts to European software houses.

A simplified view of the interconnection of the various subsystems can be obtained by examining the information flow through the computer system. Data are received at the 12 m diameter antenna in the Odenwald and transmitted over land lines through the data transmission and routing system.

Image acceptance and conditioning take place in the Siemens front-end processors. The images are then transferred to an ICL 2980 where the main processing tasks are carried out with some help in rectification and correlation from Rovsing array processors and hard-wired fast Fourier transform units. Each image line is processed in real time at the rate of one line every 0·6 s and stored on intermediate files. Data are then available for inspection by the meteorologists using the MIEC subsystem videos. Data are then archived and formatted data are assembled for distribution through the global telecommunication service and scheduled for transmission through the Siemens back-end processors to the antenna at a rate of one image format every 4 min. In parallel with this product data, satellite-status information is being received at the Meteosat operations control centre subsystem and control data are being transmitted automatically on instruction of the operations controllers.

In practice the interconnection is more complex, as most equipment is duplicated and is capable of being switched between mainframes. The system was designed to

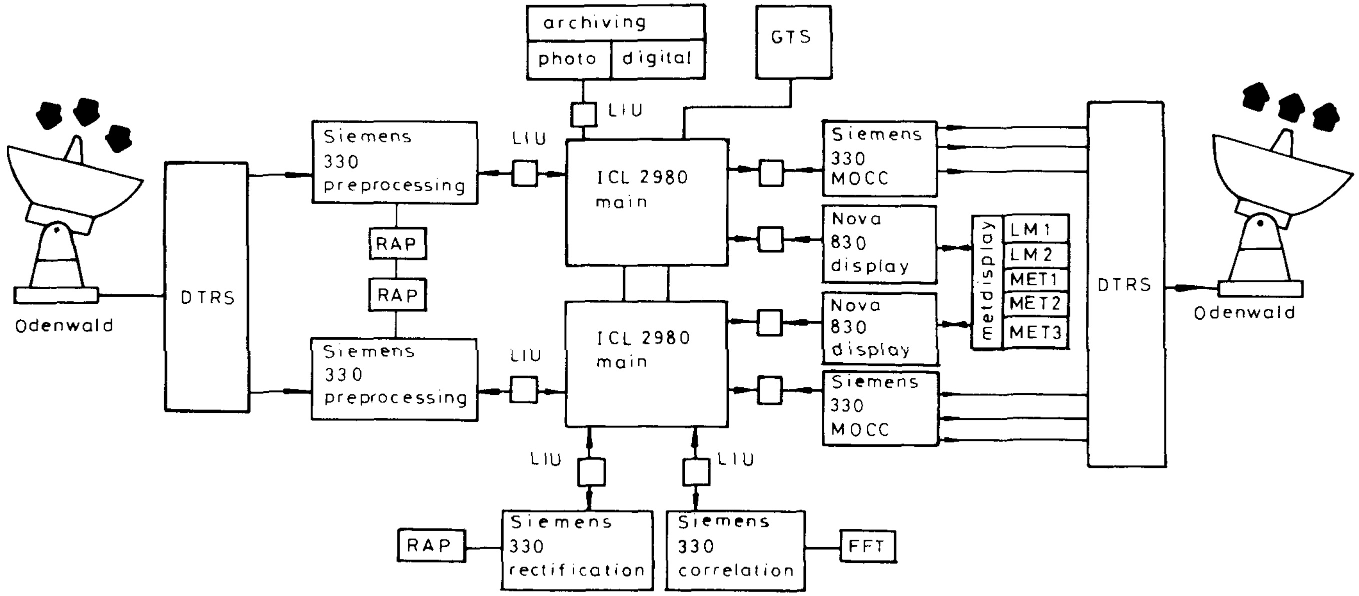


Fig. 7 Ground control hardware

be capable of running in partial hot standby mode. During the 500 h acceptance trial in September 1978 the back-end provided a hardware availability of 99•72% for spacecraft control. The total system provided a hardware availability of 99·3% for essential data acceptance, preservation functions and dissemination, and 97·2% for image and meteorological processing.

The criteria set by the Agency for these tasks were, respectively, 99·7%, 99%, and 95%. Since the trial the back-up machine has been used primarily for application development work and is only being used in cold standby giving an average hardware system availability for image processing of 96% 24 hours per day, 7 days per week. End-user availability of the real-time dissemination products varies from 85 to 95% and for specific products such as wind determinations is 95% and above.

5 Operating system

To provide the facilities described, ICL has enhanced the 2900 VME/K operating system and has provided 'hooks' to enable ESA to interpose its own system software at the operating system user interface. VME/K-ESA contains several major

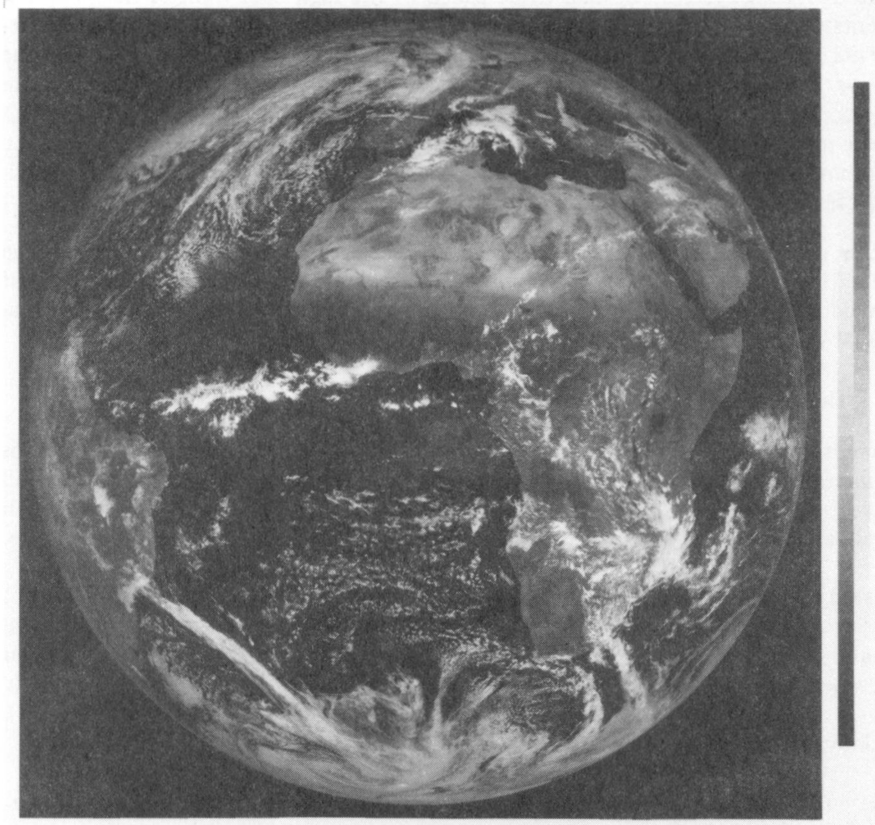

METEOSAT
1979 MONTH 3 DAY 2 TIME 1155 GMT (NORTH) CH. VIS 2
NOMINAL SCAN/PREPROCESSED SLOT 24 CATALOGUE 1018010204

Fig. 8

additional facilities. To communicate with the non-ICL processors a high-speed communications protocol had to be developed. Although designed in 1974 the resulting protocol is similar in structure to X-25. To provide for the high level of availability, the software, including applications, had to be capable of being switched from one mainframe to another in 3 min. Peripheral controllers are provided with automatic recovery features which allow recovery from most transient hardware failures. If a fault is unrecoverable then VME/K-ESA makes the controller unavailable to the system and continues working without it. An alternative controller can then be switched in by operator command.

The standard VME/K product, since it was designed originally for ICL medium range computer systems, contained a fairly simple scheduler. The Agency has provided a higher level scheduler which enables over 1000 jobs to be scheduled per day on a time critical basis. User jobs operate at a concurrency level of 22-25 in a 3·75 Mbyte 2980 during the peak periods of meteorological processing. In addition there are also four spooling concurrencies and 2 MAC jobs in the operational system.

The operating system also had to be tuned to the high data transfer flow requirements of 166 kbit of information per 0·6 s and to the strenuous real-time requirements of processing the information within 0·6 s every 0·6 s while sustaining the above multiprogramming levels. Some of the early application programs were developed on an ICL System 4/72 computer under Multijob also installed at ESOC. As a result a requirement existed for a transfer of development jobs between the System 4 and 2980. Hence one of the JCL/Macro systems available in VME/K-ESA is familiar to System 4 users.

After the early teething troubles always associated with the development of large operating systems VME/K-ESA is maturing as a product. In the spring of 1979 it provided a mean free time between software failures of 218 hours on the operational system and 100 hours on the development system.

6 The future

Meteosat 1 is exceeding expectations in performance. So when Meteosat 2 is launched in 1980 the operational software in the ground computer system will have been modified to provide for the operational control of two satellites in orbit at the same time. Full processing will continue for one spacecraft at a time.

Already in 1979 the European meteorological community is preparing plans for two further satellites as Meteosat moves from an experimental to operational phase. Future Meteosats are planned to be operational satellites providing weather data between 1983 and 1993.

Acknowledgment

The author wishes to express his thanks to the Meteosat Data Management Department for their help in the preparation of this paper. Further information on aspects of the system can be obtained from: Meteosat Data Management Department, ESOC, Robert-Bosch-Str. 5, D - 6100 Darmstadt, Federal Republic of Germany.

An analysis of checkpointing

Alan Brock
Senior Consultant Lecturer, ICL Education and Training Centre
Beaumont, Old Windsor, Berkshire

Abstract

Checkpointing is a technique that is often used to reduce the penalties incurred when, because of some failure, normal execution of a program cannot continue; the technique allows the program to be resumed correctly from an earlier point in its execution. This paper presents a quantitative analysis of the effects of checkpointing, giving:

(*a*) estimates of the optimum time interval between checkpoints
(*b*) estimates of the penalties incurred by including checkpointing in programs and restarting a program after some failure (i.e. the penalties incurred because of the fallibility of computers, data and programs)
(*c*) comparisons of those penalties with those incurred by *not* using checkpointing.

The paper is oriented primarily towards batch programs.

1 The checkpointing technique

While a program is being obeyed, a failure may occur that prevents the program from continuing normal execution to successful completion. Such failures can be caused by, *inter alia*:

(*a*) failure of the computer system (hardware, operating system etc.)
(*b*) erroneous data, if the program does not include adequate error-handling procedures
(*c*) deliberate or accidental termination of the program run by the computer operators
(*d*) errors in the program itself, such as attempted division by zero or looping indefinitely. (These may be consequences of errors in the data.)

Failures of type (*d*) can often be regarded as irrecoverable, since they may necessitate correction of the program. Any attempt to rerun the program without correcting the error may merely result in recurrence of the failure. Many other failures, including those of types (*a*) and (*c*), are often recoverable in the sense that, if the program is rerun as it stands, the failure may not recur and the program may run to successful completion. Failures of type (*b*) may also be recoverable by correcting the data and rerunning the program.

If a program makes no provision for restarting part-way through a run (e.g. by checkpointing), then recovery from a recoverable failure necessitates starting a fresh run from the beginning. The work that was done during the abortive run is then repeated and is therefore effectively wasted.

Checkpointing allows a failed run of a program to be resumed from an earlier point in its execution, and not necessarily from the beginning, with obvious benefit by reducing the amount of repeated, and therefore wasted, work. Checkpointing is normally achieved by writing, at intervals during the run, appropriate information to a checkpoint file, which is usually a serial file. This information will include:

(*a*) all or part of the program's store image, especially data areas that may be altered during execution
(*b*) the settings of relevant registers, accumulators etc.
(*c*) a note of the current positions reached in the files being accessed by the program.

If a failure occurs, the state of the program as it was at the time of a checkpoint can be recreated by:

(*a*) reconstituting the program's store image
(*b*) resetting the relevant registers, accumulators etc
(*c*) repositioning the files.

The program's execution can then be resumed, effectively as though the time between making the checkpoint and the instant of the failure had not existed. The work done in that interval of time is repeated (and thus wasted), but that is normally much less than would be wasted if a fresh run were initiated.

The making of a checkpoint constitutes work (by the processor and peripherals) that is of no direct value if no failure occurs to necessitate a restart from that checkpoint and that would not be necessary if the computer, data etc. were infallible. In short, it is a true overhead and it is desirable to minimise it. This argues for the minimum number of checkpoints, and therefore for a long time interval between checkpoints. On the other hand, if a failure does occur, restarting involves both the work to restore the correct state of the program and (usually more significantly) the repetition of work done since the checkpoint was made. This argues for a short time interval between checkpoints. Clearly one can expect to find an optimum balance between these opposing requirements. One purpose of this paper is to investigate that optimum.

The analysis in this paper is based on the premise that run time is the prime criterion and that the objective is to minimise the total run time consumed in achieving a given number of complete, successful runs of a given program. In a single-programming computer, run time is indeed the most important measure of the consumption of machine resources. In a multiprogramming or multi-virtual-machine computer, other considerations such as store occupancy and the amount of processor and peripheral activity are also important and are all likely to be increased by the use of checkpointing. They must therefore be taken into account in assessing the practical benefit of checkpointing.

2 The analysis

There are two cases to be considered, programs without checkpointing and programs with checkpointing.

2.1 Programs without checkpointing

Consider a program which, if it runs without failure, reaches successful completion after an elapsed time of r. It is subject to recoverable failures occurring randomly in time, with a mean time interval between failures of f. Then the interval t between successive failures can be assumed to follow a negative-exponential distribution with the probability density function:

$$p(t) = \frac{1}{f}\exp(-t/f)$$

Assume, for simplicity, that a run of the program is initiated at the start of an interval between failures. Then the probability that the program will run to successful completion is:

$$\Pr(\text{success}) = \Pr(t \geqslant r) = \int_r^\infty \frac{1}{f}\exp(-t/f)\,dt$$

$$= \exp(-r/f) \qquad (1)$$

It follows that the mean number of runs needed to achieve a complete, successful run is

$$M = \exp(r/f) \qquad (2)$$

and

$$\Pr(\text{failure}) = 1 - \exp(-r/f) \qquad (3)$$

Consider the time interval t to $t + dt$ where $t + dt < r$. Out of a large number of runs, the proportion $p(t)\ dt$ will fail in that interval after a mean run time of $t + ½\ dt$. Thus the mean duration of an unsuccessful run, if terms of order $(dt)^2$ are ignored, is

$$T_{av_F} = \int_0^r \frac{t}{f}\exp(-t/f)\,dt = f\left\{1 - \left(1 + \frac{r}{f}\right)\exp(-r/f)\right\} \qquad (4)$$

Out of the same large number of runs, the proportion $\exp(-r/f)$ will succeed, each having a duration of r. Therefore the average duration of all runs is

$$T_{av} = f\left\{ 1-(1+\frac{r}{f})\exp(-r/f)\right\} \left\{1-\exp(-r/f)\right\} + r\exp(-r/f)$$

whence

$$\frac{T_{av}}{r} = \frac{f}{r}\left\{1-2\exp(-r/f)\right\} + (1+\frac{f}{r})\exp(-2r/f) \qquad (5)$$

The *total* run time needed on average to achieve one complete, successful run is therefore:

$$\hat{T}_{av} = M\,T_{av} \qquad (6)$$

whence

$$\frac{\hat{T}_{av}}{r} = \frac{f}{r}\left\{\exp(r/f) - 2\right\} + (1+\frac{f}{r})\exp(-r/f) \qquad (7)$$

Note the following:

(*a*) As $r/f \to 0$, $T_{av}/r \to 1$ from below and $\hat{T}_{av}/r \to 1$ from above.

This accords with expectation since, if the time for a complete run is small compared with the mean interval between failures, most runs will be complete, that is, most run times will equal r. The few runs that fail will have shorter elapsed times and therefore the average time for all runs will be slightly less than r. Because most runs succeed, there will be little time lost on abortive runs and therefore the total run time needed to achieve one complete run will, on average, be only slightly greater than r.

(*b*) As $r/f \to \infty$, $T_{av}/r \to f/r$ and $\hat{T}_{av}/r \to \infty$

These, too, are what one would expect. When the time for a complete run is large compared with the mean interval between failures, most runs will fail, their average elapsed time being f, i.e. $\hat{T}_{av} \to f$. In the same circumstances, the success rate will be so small, and the time wasted on abortive runs so large, that the total time expended for each successful run will be large, i.e. $\hat{T}_{av} \to \infty$.

Note also that, in the absence of checkpointing, the performance depends on the single parameter r/f. It will be shown later that, with checkpointing, the relative performance is independent of the run time.

2.2 *Programs with checkpointing*

As described earlier, if a program includes checkpointing, it periodically writes selected information to a file. Doing so takes time and processor and peripheral activity, thereby increasing the elapsed time for a run, even if no failure occurs. If the extra elapsed time taken to create each checkpoint is t_w, then the total elapsed time for a run is, with n checkpoints,

$$r + nt_w \tag{8}$$

If the time interval between checkpoints is T (assumed constant), then the number of checkpoints is

$$n = \frac{r}{T} \tag{9}$$

In practical terms, there must be an integral number of checkpoints during a run. Thus, strictly, n should be truncated to the nearest integer below. In the following analysis that constraint is ignored. Note also that, in calculating n, only the basic run time is considered, the extra elapsed time due to checkpointing being ignored in computing the number of checkpoints; i.e. T is the interval between the completion of one checkpoint and the commencement of the next. In practice the taking of checkpoints is often directly geared to the application-oriented functions of the program (for example, by checkpointing every so many records) and is therefore consistent with this definition of T. Thus the time per run without failures is

$$r + \frac{r}{T}t_w = r\left(1 + \frac{t_w}{T}\right) \tag{10}$$

Consider an arbitrarily large number N of runs. The expected number of failures is

$$\frac{\text{total run time}}{\text{mean interval between failures}} = \frac{Nr}{f}\left(1 + \frac{t_w}{T}\right) \tag{11}$$

If we assume that T is small compared with f (as is likely to be the case in practice and is justified *a posteriori* for the optimum interval) then, on average, any failure that occurs will do so midway in an interval between successive checkpoints. Therefore work occupying $T/2$ units of elapsed time will need to be repeated. Suppose that each such restart also involves t_r units of elapsed time to recreate the program's state from a checkpoint. Then each restart/repeat incurs a total of $t_r + ½T$ units of elapsed time or, from eqn. 11,

$$\frac{Nr}{f}\left(1 + \frac{t_w}{T}\right)\left(t_r + ½T\right) \tag{12}$$

in total for N runs of the program. But a failure may occur during the restart/repeat actions, the expected number of such failures being

$$\frac{Nr}{f^2}\left(1 + \frac{t_w}{T}\right)\left(t_r + ½T\right)$$

and each can be assumed, as an approximation, to incur a further time of $t_r + ½T$. Thus the total extra time for these 'failures during recovery' is

$$\frac{Nr}{f^2}\left(1 + \frac{t_w}{T}\right)\left(t_r + ½T\right)^2$$

But failures may occur during that extra time and so on, whence the total time involved in restart/repeat actions is:

$$Nr\left(1+\frac{t_w}{T}\right)\sum_{j=1}^{\infty}\left(\frac{2t_r+T}{2f}\right)^j = Nr\left(1+\frac{t_w}{T}\right)\left(\frac{2t_r+T}{2f-2t_r-T}\right) \tag{13}$$

Hence the total time for all N runs is, from eqns. 10 and 13

$$Nr\left(1+\frac{t_w}{T}\right)\left\{\frac{2f}{2(f-t_r)-T}\right\} \tag{14}$$

and the mean run time taken over all runs is

$$r\left(1+\frac{t_w}{T}\right)\left\{\frac{2f}{2(f-t_r)-T}\right\} = Fr\text{, say} \tag{14a}$$

w(here

$$F=\left(1+\frac{t_w}{T}\right)\left\{\frac{2f}{2(f-t_r)-T}\right\} \tag{14b}$$

In eqn. 14*b* the first term on the right-hand side represents, essentially, the time consumed in writing checkpoints and the second is related to the time taken in recovery from failures.

Given that the objective is to minimise the average run time, it is necessary to find the minimum value of F, F_{min}. For a given program with a given form of checkpointing running on a given computer, r, f, t_w and t_r are fixed, leaving T as the only free variable in eqn. 14*b*. Thus to minimise F, T must have the value, T_{opt} say, that gives $dF/dT = 0$.

Now

$$\frac{dF}{dT}=\left(1+\frac{t_w}{T}\right)\left\{\frac{2f}{(2f-2t_r-T)^2}\right\}-\frac{2f}{2(f-t_r)-T}\left(\frac{t_w}{T^2}\right)$$

whence

$$T_{opt}=\surd\left[t_w\left\{2(f-t_r)+t_w\right\}\right]-t_w \tag{15}$$

Note that T_{opt} and F are independent of r.

In many practical cases, t_w and t_r are small compared with f, e.g. seconds or fractions of a second compared with hours. In such cases approximations can be made, as follows:

$$T_{opt}\triangleq\surd(2t_wf) \tag{15a}$$

$$F_{min}\triangleq(1+a)/(1-a)\triangleq 1+2a \tag{14c}$$

where $a = \sqrt{(t_w/2f)} \simeq T_{opt}/2f$ is small and terms of order a^2 are neglected.

These approximations over-estimate T_{opt} and underestimate F_{min}.

2.2.1 *Discussion of t_w and t_r*

t_w is the elapsed time to write a checkpoint, t_r is the time to recreate the state of the program as it was at the time of a checkpoint. Thus t_w includes the times to:

(*a*) locate the place in the checkpoint file at which to write the checkpoint information
(*b*) write that information
(*c*) return to execution of the 'application-oriented' part of the program.

and possibly the time to bring in the checkpoint-writing routine from an overlay or virtual store.

Of these, (*a*) will depend on the medium used for the checkpoint file. If it is a magnetic tape, used only for the checkpoint file, then the time will be zero. If it is a disc used only for the checkpoint file the time will consist of rotational delay (typically 0 to 25 ms). If the disc is shared with other programs the time is likely to be that for a random access, including head movement and queuing delays, perhaps 50 to 200 ms.

Time (*b*) is the time to write the appropriate amount of data to a serial file. If the file is on a disc shared with other programs the writing may become, in effect, random writing, with head movements and queuing delays between successive blocks.

Time (*c*) will usually consist only of the processing time to return control from the checkpointing procedure to the 'main' program and is thus negligible.

The restart time t_r includes the times to:

(*a*) locate the required checkpoint data in the checkpoint file
(*b*) read that data
(*c*) reposition the other data files
(*d*) return to execution of the 'application-oriented' part of the program.

Of these, (*a*) depends on the medium used for the checkpoint file, and on the strategy used to locate the correct block(s). For example, if the checkpoint file is on magnetic tape it may involve rewinding the tape to the beginning and doing a serial search. At the other extreme it may involve merely a 'skip back to tape mark' and read, or even 'read reverse' a few blocks. On disc this time will be similar to (*a*) of t_w.

Time (*b*) is the time to read the relevant checkpoint data, and can be assumed to be equal to time (*b*) of t_w.

Time (*c*) also depends on the medium and the facilities available for repositioning. For disc files it can be regarded as the time for a random positioning on each file, overlapped with each other if the computer system permits. For magnetic-tape files it may involve rewinding and serially searching each file, or it may involve skipping to tape marks if such marks are written to the data files whenever a checkpoint is made.

If a data file is on a disc and is updated *in situ* (ie the updated version of a record overwrites the previous version), then mere physical repositioning of the heads may not be sufficient restart action. Instead it may be necessary to reverse the updating that has been done since the checkpoint was made, by reference to associated recovery files. Such actions may take some minutes.

The repositioning of input files on slow media such as cards and paper tape is likely to necessitate operator action, the time taken being difficult to assess in general terms. For slow output devices such as line printers no physical repositioning may be necessary in fact, since any repeated output can usually be excised manually after completion of the run, taking no computer time directly. However, auditing and security requirements may call for stricter control relating to repeated output.

Time (*d*) is usually negligible.

3 Examples

Example 1

A program has a basic run time r of 1 h and is subject to failures at random with a mean interval of 8 h. Checkpointing involves writing 50,000 bytes to magnetic tape with an effective transfer rate of 200 kbyte/s. The magnetic tapes have a 'read reverse' facility. The four data files are on dedicated disc cartridges with an average time for random access of 50 ms.

What are:

(*a*) the optimum inverval between checkpoints T_{opt}
(*b*) the average time per run with checkpointing Fr
(*c*) the average time per run without checkpointing T_{av} ?
(*d*) the average total time per successful run without checkpointing $\hat{T}_{av}$?

First calculate t_w:

(*a*) with checkpoint file on magnetic tape, time to position file for writing = 0
(*b*) time to write 50,000 bytes at 200 kbyte/s = 0·25 seconds
$\therefore t_w = 0{\cdot}25$ s.

For t_r

(*a*) with read reverse, time to locate checkpoint data = 0
(*b*) time to read 50,000 bytes = 0·25 s.
(*c*) time for four random positionings on the data files = 200 ms = 0·2 s.

Therefore

$$t_r = 0{\cdot}45 \text{ s.}$$

$$f = 8 \text{ h} = 28{,}800 \text{ s}$$

$$T_{opt} = \sqrt{\left[0{\cdot}25 \left\{2(28{,}800 - 0{\cdot}45) + 0{\cdot}25\right\}\right]} - 0{\cdot}25$$

$$= 119{\cdot}75 \text{ s (say 120 s)}$$

(Note how small t_r is compared with f in this case.)

With this value for T, F min becomes

$$F_{min} = \left(1 + \frac{0{\cdot}25}{120}\right)\left(\frac{57600}{57600 - 0{\cdot}9 - 120}\right) = 1{\cdot}0042$$

Thus the average run time = 1·0042 h.

Checkpointing and failures thus increase the average run time by 0·42%, i.e. 15 s in 1 h.

In the absence of checkpointing, with $r/f = 1/8$, the number of attempts needed to achieve a complete run is exp (1/8) = 1·133.

$$T_{av} = 8\left\{1 - 2\exp(-1/8)\right\} + 9\exp(-1/4) = 0{\cdot}8893$$

$$\hat{T}_{av} = T_{av}/\exp(-r/f) = 0{\cdot}8893 \times 1{\cdot}133$$

$$= 1{\cdot}0076$$

Thus if checkpointing were not used the average duration of each run would be 0·8893 h and a total of 1·0076 h would be expended on average for each useful 1 h run, i.e. failures would lead to consumption of 0·76% more elapsed time than on an infallible machine.

For this particular example it might be decided that checkpointing is not worthwhile, since it saves little time (about 12 s per hour), against which must be set the increased size of the program (and hence increased store occupancy and perhaps paging), the extra load on the processor, the extra demands on the peripherals, and the extra time and cost to write, test and develop the program.

Example 2

This assumes the same program as before, but subject to failures at a mean interval of 4 h.

Example 3

A program with a basic run time of 6 h is subject to a mean interval between failures of 12 h. Checkpointing involves writing 100,000 bytes to a shared disc in 2000 byte blocks, the average access time per block being 100 ms (allowing for queuing and contention for the disc). Two data files are on disc, with the same characteristics as in Example 1, and the other two files are multi-reel magnetic-tape files with the same characteristics as the tape in Example 1.

For writing the checkpoint, t_w can be calculated as the time to write 50 blocks at 100 ms per block, i.e. 5 s. This includes the initial positioning time.

For t_r, locating and reading the checkpoint data will take the same time as locating and writing which is 5 s.

Repositioning the disc data files will take (as Example 1) 50 ms each = 100 ms for two. For repositioning the magnetic-tape files, it is assumed here that:

(*a*) the tapes must be rewound and then wound forward for, on average, half a reel length, taking the assumed time of 3 min

(*b*) repositioning of the magnetic-tape files can be overlapped with each other, but not with repositioning of the disc files, and no repositioning can be done until the checkpoint file has been read.

Then $t_r = 5 + 3 \times 60 + 0{\cdot}1$

$$\doteq 185{\cdot}1 \text{ s}$$

$$T_{opt} = \surd\left[5\left\{2\,(43200 - 185{\cdot}1) + 5\right\}\right] - 5$$

$= 651$ secs and $F_{min} = 1{\cdot}0197$

Therefore average run time = $6 \times 1{\cdot}0197$ h = 6h 7 min

$r/f = 0{\cdot}5, f/r = 2$

$$\frac{T_{av}}{r} = 0{\cdot}6775$$

Therefore the average run time without checkpointing is

$6 \times 0{\cdot}6775$ h = 4 h 4 min

$$\frac{\hat{T}_{av}}{r} = 0{\cdot}6775/\exp\,(-0{\cdot}5) = 1{\cdot}1170$$

Average time expended per complete run = $6 \times 1{\cdot}117$ h

= 6 h 42 min

Number of attempts per complete run = exp (0·5) = 1·65

In this case, then, checkpointing offers a saving of some 35 min compared with no checkpointing, and without checkpointing it would be necessary on average to make 1·65 attempts to achieve each successful run, i.e. 39% of the runs initiated would fail.

4 Results

Some results from the analysis are presented in Figs. 1-5.

Fig. 1 shows the variation of M, T_{av}/r and T_{av}/r with r/f for a program without checkpointing. It demonstrates clearly the increase in the number of attempts M necessary to achieve a successful run as the basic run time r increases in relation to the mean interval between failures f, and the corresponding *decrease* in average run time T_{av}/r and the *increase* in the total elapsed time $(\hat{T}_{av}/r)$ consumed in achieving a successful run. The decrease in average run time (T_{av}) comes about, of course, because an increasing proportion of runs fail before completion, i.e. with actual run time less than basic run time. From eqn. 4, it can be shown that, if only runs that fail are considered, T_{avF}/r is zero at $r/f = 0$ and ∞, and has a maximum of 0·298 when $r/f = 1·79$.

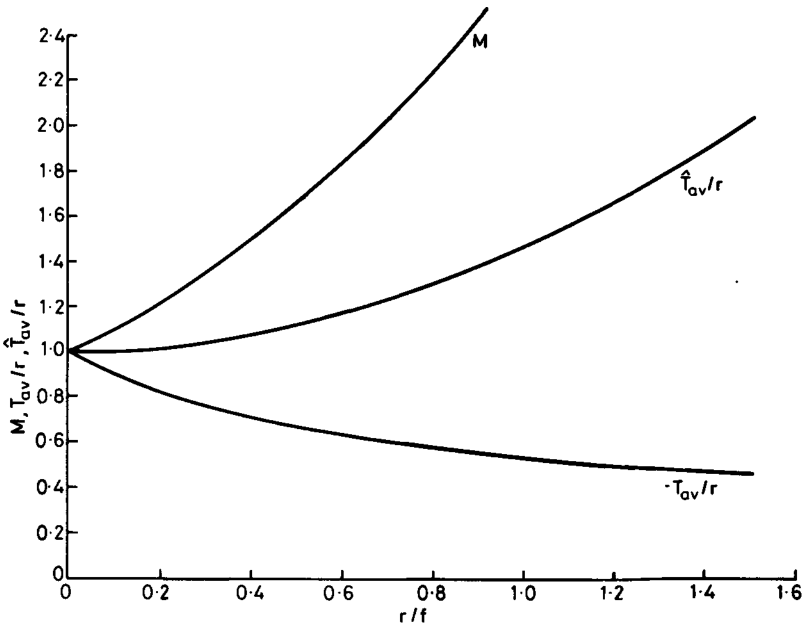


Fig. 1 Programs without checkpointing. Variation of M, T_{av}/r and $\hat{T}_{av}/r$ with r/f

If we turn now to programs that do use checkpointing, Fig. 2 shows how T_{opt}, the optimum interval between checkpoints, varies with f, the mean interval between failures. Curves are shown for four combinations of t_w (time to make a checkpoint) and t_r (time to restart from a checkpoint). The curves show clearly that t_w has a marked effect on T_{opt} while t_r has relatively little effect, such effect as it has decreasing as f increases. These results are of course readily seen from the form of eqn. 15, as also is the fact that, if t_w and t_r are both small compared with f, then T_{opt} is approximately proportional to $\surd f$ as is shown by the nearly straight curves with slope 0·5 in Fig. 2 and by the curve of $\surd(2t_w f)$ for $t_w = 250$ s.

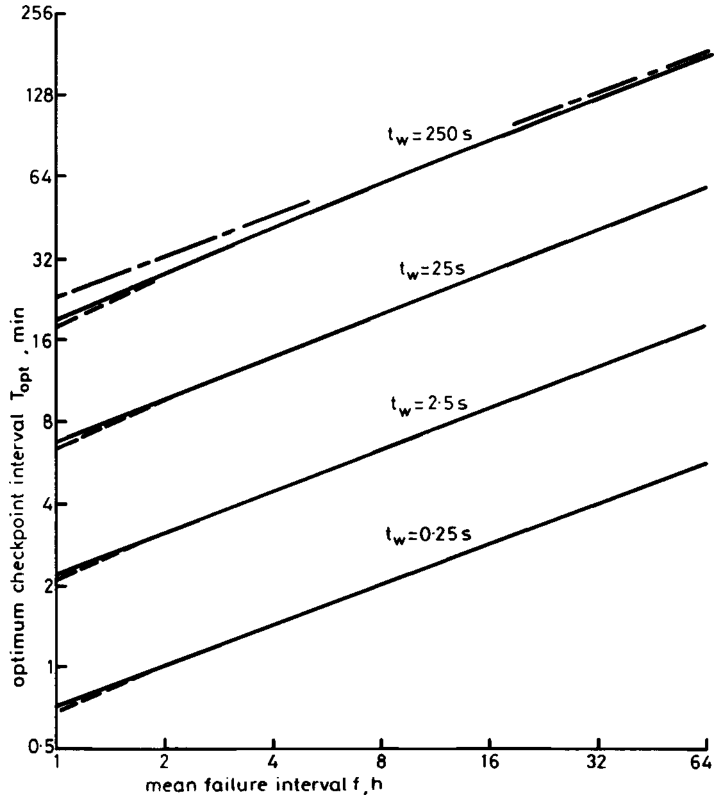


Fig. 2 Variation of optimum checkpoint interval with mean failure interval

t_w as stated, t_r = 0·25 s
t_w as stated, t_r = 250 s
$T_{opt} = \surd(2t_w f)$ for t_w = 250 s

If we take the particular case of $t_w = t_r = 2{\cdot}5$ s, the range of T_{opt} is interesting, perhaps even surprising. When $f = 1{\cdot}0$ h (an extremely high failure rate) T_{opt} is just over 2 min, while if $f = 64$ h, T_{opt} is about 18 min. Perhaps because some manufacturers' contracts exclude liability for more than 20 min spoiled work, there appears to be a tendency to checkpoint at 20 min intervals: the present analysis shows this rule-of-thumb to be substantially away from the optimum in many practical cases.

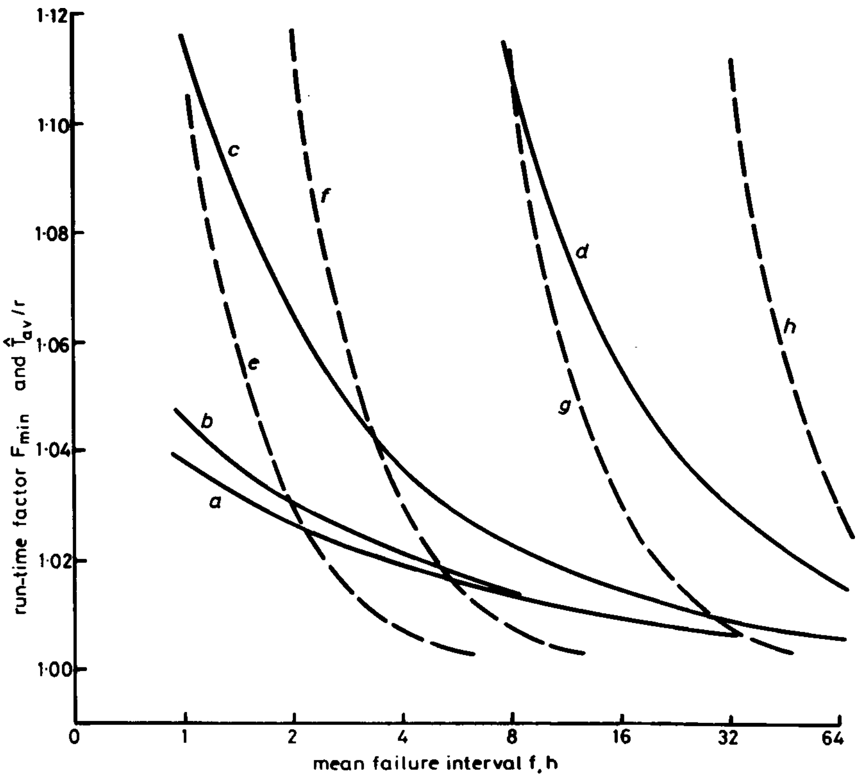


Fig. 3 Increase in run time with and without checkpointing

	with checkpointing		without checkpointing
	$t_w = 2{\cdot}5$ s	(e)	$r = 0{\cdot}5$ h
(a)	$t_r = 0{\cdot}25$ s and 2·5 s	(f)	$r = 1$ h
(b)	$t_r = 25$ s	(g)	$r = 4$ h
(c)	$t_r = 250$s	(h)	$r = 16$ hr
(d)	$t_r = 2500$ s		

The solid curves in Figs. 3 and 4 show how F_{min} varies with t_w, t_r and f. In Fig. 3, t_w is constant at 2·5 s and in Fig. 4, t_r is constant at 25 s. It is seen that F_{min} increases as t_w and t_r increase and decreases as f increases. Qualitatively, these variations are what one would expect intuitively; Figs. 3 and 4 show the quantitative dependencies. Also shown in Figs. 3 and 4, as the broken curves, are the variations of $\hat{T}_{av}/r$ with f for various values of the basic run time r, calculated from eqn. 7. Now, both F_{min} and $\hat{T}_{av}/r$ are factors by which the basic run time of a program is increased because of fallibility; they are therefore directly comparable quantities. In particular, if, for any given case, $F_{min} < \hat{T}_{av}/r$ it follows that the use of checkpointing is beneficial from the point of view of machine time consumed in achieving a given number of successful runs of a program. If $F_{min} > \hat{T}_{av}/r$, less machine time is consumed without checkpointing than with. Suppose, then, that a program has a basic run time r of 4 h and, if it is checkpointed, t_w = 2·5 s and t_r = 250 s. Then, from curves (c) and (g) in Fig. 3, it is seen that checkpointing is beneficial if f is less than about 30 h, and not beneficial if f exceeds that value.

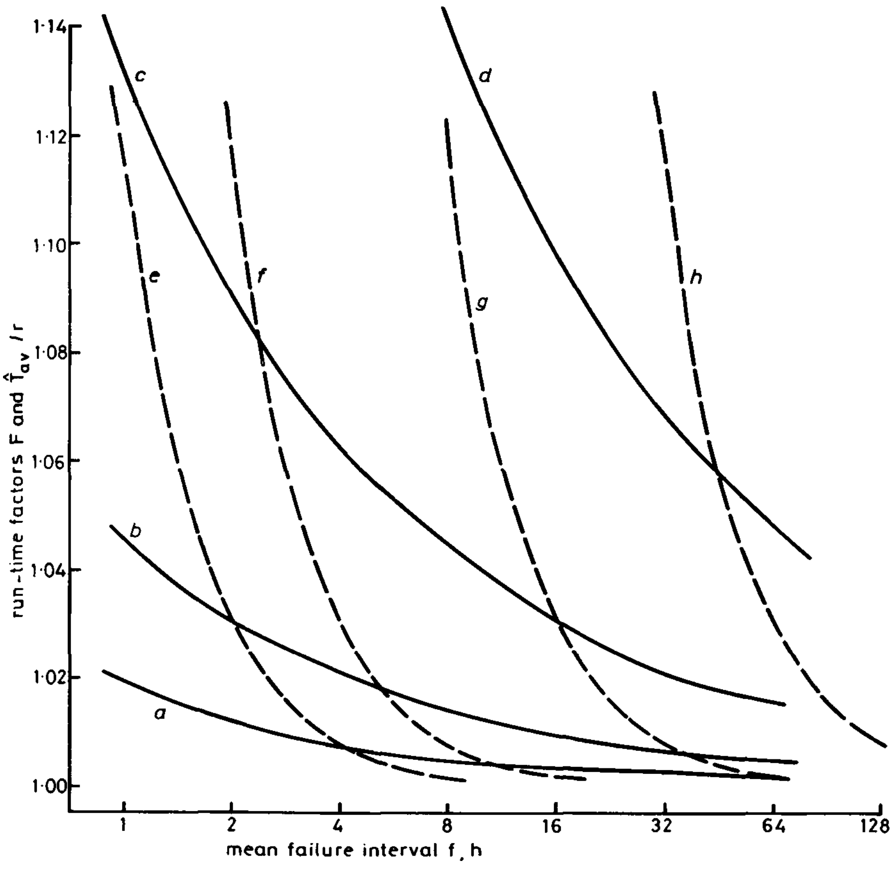


Fig. 4 Increase in run time with and without checkpointing

with checkpointing		without checkpointing	
	t_r = 25 s	(e)	r = 0·5 h
		(f)	r = 1 h
(a)	t_w = 0·25 s	(g)	r = 4 h
(b)	t_w = 2·5 s	(h)	r = 16 h
(c)	t_w = 25 s		
(d)	t_w = 250s		

Given that there is, for any given case, an optimum interval between checkpoints, what is the effect of checkpointing at some other interval? Fig. 5 shows the effect for the particular case $t_w = 25$ s, $t_r = 250$ s; F is plotted against T, the actual interval between checkpoints, for various values of f, the mean interval between failures.

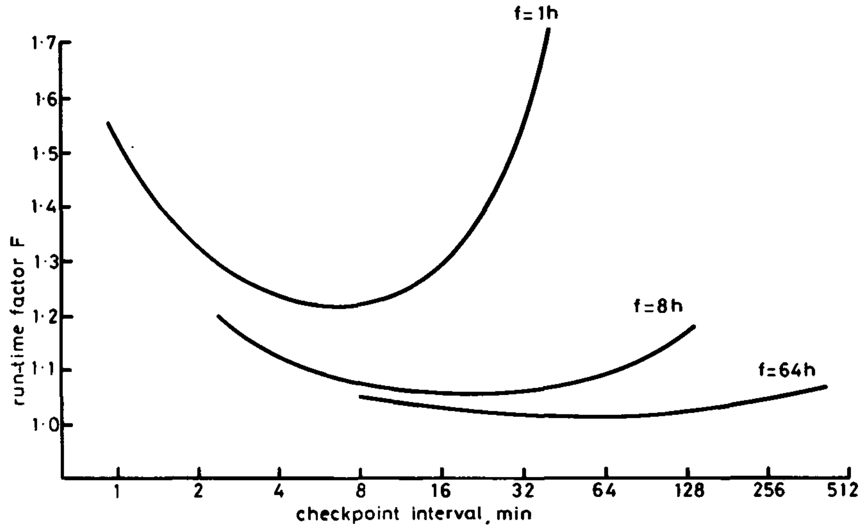


Fig. 5 Effect on run time of varying the checkpoint interval
$t_w = 25$ s, $t_r = 250$ s

For $f = 1$ h, $t_w = 25$ s, and $t_r = 250$ s the effects of checkpointing at half and at twice the optimum interval are given in Table 2. It is seen that:

(*a*) departing from the optimum interval by a factor of 2 in either direction results in a 3% increase in run time

(*b*) making the interval twice the optimum causes the greater increase in F.

Fig. 5 and other graphs not reproduced here show that the effects of departure from the optimum checkpointing interval decrease as f increases and increase as t_w and t_r increase.

Table 2

T	F	F/F_{min}
s		
192·5	1·250	1·030
385	1·214	1
770	1·254	1·032

A program with checkpointing is likely to be more *consistent* in its time to completion than is a program without checkpointing, even though the *average* elapsed time to completion may actually be longer than without checkpointing. This consistency can be advantageous *vis-a-vis* scheduling of programs, especially if there is a deadline for completon of the run.

Another point of practical importance is operational convenience. Consider a program that updates multireel files on magnetic tape. Without checkpointing, recovery from a recoverable failure would necessitate remounting and reprocessing all the reels that had been processed up to the time of the failure. With checkpointing, it would be necessary to remount only those reels that had been processed since the most recent checkpoint; this might well involve no tape changing at all for the restart.

Further, as this paper has shown, F is not very sensitive to deviations from T_{opt} for many practical values of f, t_r and t_w. That being so, it may cause little increase in F, and yet make operation of the program much simpler and quicker, if checkpointing is arranged to coincide with the natural changing of tape reels. If that is done it may be possible to write the checkpoint data at the front of each main-file reel, eliminating the need for a separate checkpoint file and easing operation still further.

5 Conclusions

(*a*) If checkpointing is used in a program, there is a particular time interval between checkpoints that minimises the average total run time consumed in achieving a successful run to completion of that program. This optimum time interval is a function of:

t_w : the time taken to write a checkpoint

t_r : the time taken to restart from a checkpoint

and f : the mean interval between failures
and is independent of the basic run time of the program.

(*b*) Depending on t_w, t_r, f and r (the basic run time of the program), the use of checkpointing may or may not be beneficial. It is beneficial if F (from eqn. 14*b*) is less than T_{av}/r (from eqn. 7).

(*c*) The criticality of conforming to the optimum interval between checkpoints varies substantially. It becomes more critical as t_w and t_r increase, and less critical as f increases.

(*d*) In deciding whether to apply checkpointing, factors such as the cost and time of program writing and development and the effect on processor and peripheral utilisation in a multiprogramming environment must be taken into account, as well as the elapsed time discussed in this paper.

Statistical and related systems

B.E. Cooper
Consultant in Statistics and Data Management
Dataskil Ltd., * Reading

Abstract

The paper discusses the design considerations that underlie the development of application systems for statistics and related subjects. There are two primary considerations. The first is the need for extreme flexibility in the interface between the user and the system. The second is the relationship between one system and another and between the systems and a database. The practical realisation of these principles is illustrated by descriptions of the main features of the ICL Applications Control Language, the ICL Data-Analysis System and the statistical system Package-X developed by Dataskil for the British government.

1 Introduction

Potential users of an application system need to be convinced that it will meet their requirements and that they will be able to use it without any fundamental change to the way in which they normally work. In most cases it is not difficult for a user to determine what facilities the system offers and whether or not it will meet his requirements. The second consideration is, however, one to which the designer must give careful consideration. He must see his system from the users' viewpoint. He must be able to persuade potential users that it is easy to learn and that it will not distract them from their main activities. This is particularly true for systems written for the data analyst, including systems for statistics, forecasting and econometrics, survey and census analysis, matrix handling and mathematical programming. There is a particularly important need in such systems for a flexible approach to the handling of data as well as for the provision of appropriate analysis facilities.

Various aspects of application systems designed for the analysis of data are discussed. The discussion begins by considering the facilities that are required and identifies those that are common to all application areas. The nature of the interface between the user and the system are considered next and the flexible approach taken by the statistics system Package-X, in which the user chooses his own level of communication, is described. The structure of the 2900 Data Analysis System embracing a number of application systems is then discussed. The final section is devoted to the possibilities for linking such application systems to a database.

*Dataskil Ltd. is a wholly-owned subsidiary company of ICL. The address is Reading Bridge House, Reading RG1 8PN

2 Facilities and properties of an application system

The development of the 2900 range provided the opportunity to consider from first principles the facilities that should be provided in application systems for the data analyst. This general class includes the aforementioned systems, i.e. those for statistics, forecasting and econometrics, survey and census analysis, matrix handling and mathematical programming. The opportunity was almost a unique one, permitting not only detailed consideration of the requirements of each application but also the study of the relationships between them.

Design studies for these individual systems quickly revealed a large area of common ground. In a previous paper[1] I discussed in some detail the facilities that are required in such systems and identified the extent and nature of the comon ground. Thus, in the Sections which follow, I confine myself to a brief statement of facilities. The identification of which facilities are particular to one, or perhaps two, application areas and which are of a general nature and useful in all applications forms the key part of the design of such systems.

2.1 Data structures and types

The design study for each application system must identify the data structures that the system is to recognise and how the various facilities it provides are to be related to these structures. Bringing together the design studies for the various applications systems for the 2900 range established that general access was required to the following data structures.

scalars	-	single value
arrays	-	many values, one or more dimensions
lists	-	list of values referred to by one name.

These structures were required to hold values of the following types:

Integer
Character
Logical
Name
Real

Readers with an eye for mnemonics will observe that the initial letters for these five types are ICL NR. Unfortunately type Character has since been dropped because it is contained within type Name.

Particular data structures for particular applications are also required. For example, the *series* is required by the forecasting system and the *data matrix* is required by the statistics and survey systems. These are described in more detail later but it should be noted at this stage that the data types that may be stored in these structures should correspond to the data types of the structures already listed so that data transfer between the general and the particular structures is permitted.

2.2 *Classification of facilities*

The facilities of an application system may be organised into three groups. The first group contains those facilities which provide access to the analyses, displays and techniques that characterise the particular application. For example, the facility for regression analysis in a statistics system or for the Box-Jenkins method in a forecasting system would fall within this first group.

The second group contains data management facilities which are particular to an application but not directly associated with a particular technique. For example, amalgamating two data matrices to form a new data matrix for analysis in the statistics system would fall within this group.

The third group contains operations on data structures, either general or particular, which are normally found in a general purpose computing language such as Fortran. I have previously[1] used the term 'support operations' to describe this group, but in the following Sections will refer to them as 'program operations'. The group contains facilities for arithmetic, branching (IF, GOTO), Looping (DO) and input and output.

It is revealing to observe the gradual development of statistics systems. Most, in their first version, have concentrated on providing facilities for particular techniques; i.e. the initial emphasis is on providing Group 1 facilities. In subsequent releases of the system the emphasis gradually changes to include a higher and higher proportion of facilities of Groups 2 and 3. The inflexibility of a system concentrating on Group 1 facilities is quickly revealed by users. Statistical analysis, for example, is usually a multistage process; it is rare for the statistician to be satisfied with just one formal analysis. As a minimum he will usually need to see some additional results or to observe the effect of modifying his original approach in some way.

Response to these pressures is therefore necessary. However, introduction on a gradual basis often produces a rather piecemeal range of facilities. This is particularly true of facilities in Group 3. Careful consideration of this group as part of the *original* design of the system is very important. When considered from the outset, the facilities in this group are found to correspond to those of a general programming language with certain concessions to the needs of the applications. In fact the facilities involving the general data structures arrays and scalars correspond very closely to those in a general computing language such as Fortran. Thus the initial task is one of designing a general computing language which can also form a natural basis for corresponding operations on the particular data structures recognised by the application systems and which can co-ordinate appropriately with the Group 1 and 2 facilities. Faced in this way the facilities fall into place in a natural manner. It will be seen in Section 4 that the design of the 2900 Data Analysis System was approached in this way and that a selfcontained language, referred to as the Applications Control Language, was produced which may be used with a number of applications systems.

2.3 Relationships between facility groups

The relationships between the facilities in the three groups is a particularly important design consideration. The need for common data types has been mentioned already as essential to ensure proper transfer of data between data structures both general and particular. A further stage in this process is to ensure proper integration of the general data structures arrays, scalars and lists with the Group 1 and 2 facilities which depend on the particular application. For example, it should be possible to include the names of scalars or arrays in statements selecting statistical or forecasting analyses. Equally, it should be possible to store values which have been computed as part of such analyses in either general or particular data structures so that further analysis involving them is possible.

To illustrate the inclusion of names of scalars consider the statement:

```
POLYNOMIAL REGRESSION OF Y ON X DEGREE ND
```

If ND is an integer scalar this statement would fit a polynomial of degree specified by the value of ND. Thus the degree of the polynomial to be fitted could be based on a previous computation or subject to comparisons with other values. Furthermore, this statement could be contained within a DO loop controlled on the scalar ND and thereby repeated for a number of different degrees of fit.

To illustrate the storage of analysis values in system data structures consider the statements:

```
REAL ARRAY CF (0:4)
REGRESSION OF Y ON  XA  XB  XC  XD
ESTABLISH COEFFICIENTS IN CF
```

The second statement fits a regression function involving four variables. The third statement transfers the coefficients of this function to the array CF which was defined by the first statement. This process, normally referred to as feedback, is a particularly important part of a system's flexibility. It is missing from a surprisingly high number of application systems.

In summary, therefore, it is important to ensure that the facilities of all three groups are included in full, that the facilities within each group are self-consistent and that the three groups of facilities fit properly together to form a coherent whole.

3 Communication between user and systems

A system designer should be aware that there is an initial threshold to be overcome by the new user of his system. The user is not interested in computing for its own sake but in the results the system can provide. He should not be distracted from his main task of analysing data by the intricacies of the system he has to use to do this analysis. Thus the time taken to learn how to use the system should be made as small as possible. Use of the system should fit naturally into the normal work pattern of the data analyst. It will not so fit if he is expected to learn detailed and complicated rules. Minimising the time taken to learn to use a system has a similar effect on the time taken by the occasional user to relearn the system. Aspects of the interface between user and systems are discussed in the next Sections.

3.1 User language

If the data analyst can write statements which are close to the form of instructions he would give to an assistant, use of the system will fit naturally into his normal work pattern. This is possible for selection of facilities in Groups 1 and 2. Statements corresponding to facilities in these groups should be as close to normal English as possible. The sequence of statements from the 2900 Data Analysis System are in a near-English form and are readily understood with little explanation:

```
REAL ARRAY C(0:3)
READ DATA MATRIX DMA WITH VARIATES Y X1  X2  X3
   FILE MYDATA
REGRESSION OF Y ON  X1  X2  X3
ESTABLISH COEFFICIENTS IN C
```

The facilities in Group 3 cannot conveniently be expressed in terms of English-like statements. The best form of statements for this group is one close to that adopted by the general computing languages such as Fortran. For example, arithmetic is best expressed in the form of the arithmetic statements found in most general computing languages. Most languages use some form of IF statement to express conditional branching. Thus, statements for this group of facilities most conveniently resemble their counterparts in established languages. Since these are rather different in form from those for Groups 1 and 2 it is important to ensure that where they share a construct with other statements this should be expressed in a common manner. The DO statement in the 2900 Data Analysis System provides an example here. Its form is:

<*label*> : DO <*dummy scalar*> = <*list*>

The construction of the *list* in this statement follows the same rules as apply to lists included in any other statement in the system's language. The values in the list must be of the same type as the scalar but all types may be used. For example, control of a DO loop using a name scalar is particularly valuable as illustrated by the following sequence:

```
L1:  DO NY. = Y1, Y4, Y7, Y8
     REGRESSION OF NY ON X1  X2  X3  X4
     END L1
```

3.2 User selection of his interface with the system: Package-X

Package-X is a statistics and data management system designed for, and in close co-operation with, the Government Statistical Service. It is currently in regular use on over 40 installations. Apart from being a useful and well-liked system its development is important because of the new approach it takes to the interface between the user and the system. Package-X can operate in four different modes, two for terminal users and two for batch.

Modes 1 and 2 are for the terminal user. Mode 1 is a fully conversational mode in which the user responds to questions put to him by the system; he requires very little knowledge of the system before he attempts to use it. He may ask the system to introduce itself by displaying information pages explaining various features, or indexes listing its facilities. If he does not understand a question he can type HELP for explanation.

Mode 2 is an interactive mode in which the user at the terminal types statements in the system's control language, which has the easy-to-use properties argued for in the previous Sections. If he makes a mistake in a statement the system will ask him appropriate Mode 1 questions to enable him to correct the error.

The user may transfer freely between Modes 1 and 2 during a run, using Mode 2 for those parts of the system he is familiar with and Mode 1 for the rest. A further degree of flexibility is scheduled for Release 5 of Package-X, due early in 1980, concerning the selection of different versions of the file on which questions and messages are stored. The user will be able to select between short and long versions of questions and alternative message files translated into other languages could easily be made available. Thus, with transfer between message files and between modes, and with Mode 1 responses to errors in Mode 2 statements, the user may select with great flexibility his interface with the system. Further flexibility is available using the macro and program facilities described in Section 3.3.

The other two modes of operation are for the batch user. Mode 3 is a normal batch mode in which the user expresses his requirements in the same control language as in Mode 2. Mode 4 is a syntax checking mode which will normally be used to check a program before it is run in Mode 3. The use of the same control language for both batch and terminal work gives the great advantage of requiring the user to learn only a single language.

3.3 Macro and program facilities

To select operations in Group 3 the user of Package-X writes what is referred to as a program. The sequence of operations is introduced by a PROGRAM statement and terminated by an END PROGRAM statement. In Release 5 the user will be able to name a program and perform a number of actions on it. For example, programs may be saved on user or installation SAVE files and fetched and run as required. Thus, commonly-used sequences of such operations may be written once and saved for other users.

Any user may define a macro consisting of any sequence of statements in the Package-X control language. These may be saved, fetched and run in the same way as programs. Since macros and programs are made up of statements used in Modes

2, 3 and 4 but not in Mode 1 they must be written by users familiar with the system's control language. However, this does not prevent a Mode 1 user from running a macro or program. Thus the macro and program facilities represent a considerable increase of flexibility in the user's use of the system no matter which mode he uses. A further increase in flexibility will take place in Release 5 when extensive facilities for editing macros and programs will be introduced.

The facilities in Package-X for defining, saving, fetching, editing and running macros and programs, coupled with facilities described in Section 3.2 for selecting different modes of operations and different message files, represent an important advance in the way in which a user may choose how he communicates with the system.

4 Communications between application systems

The first consideration in the design studies for the various application systems for the 2900 series was the identification of those facilities common to many systems. This has been discussed in some detail in Section 2 where it was identified that the data structures scalars, arrays and lists were required in all application systems and that the facilities for their access and manipulation corresponded to those of a general computing language such as Fortran. The language which has been designed to meet this need is the *Applications Control Language.* This provides the necessary common facilities for all applications and is described briefly in Section 4.1.

The second consideration was the need to achieve flexible communication of data between applications systems. The need to transfer data freely between survey and statistics systems has been recognised for many years, but many other situations requiring interchange of data between one application and another were readily acknowledged. To meet this need the 2-tier structure of the *Data Analysis System* was devised, with the Applications Control Language forming the system's foundation. The structure of this system embracing many application areas is described in Section 4.2.

4.1 The Applications Control Language

The Applications Control Language (ACL) for the 2900 series is a selfcontained language with most of the facilities of a general computing language such as Fortan. It forms the basis of many of the 2900 series application systems and provides a means of communication between them.

ACL is a block structured language in which the outermost block must be called PROGRAM. Thus an ACL program begins with a BEGIN PROGRAM statement and closes with END PROGRAM. Other blocks may be opened as required by the user with the names of his own choice. Procedure definitions and CALL and RETURN statements of the form common in many languages are available to define and to access user procedures.

ACL recognises the data structures scalars, arrays and lists which may be of types integer, logical, name and real. Statements to declare these structures and give them initial values are similar to those of other languages; they must follow the conventions for expressing lists, in shorthand form where appropriate, which apply throughout ACL and any application systems associated with it.

Assignment statements are available to assign values to all four data types. Real, integer and logical assignment statements follow the normally accepted rules. Name assignment statements are initially confined to a restricted right hand side form consisting of a constant or a single variable of appropriate type.

The looping statement again employs the form of list construction used throughout ACL and its associated application systems. The fullest form of the loop is:

```
< label > : DO < control scalar > = < list > WHILE < logical expression >
            ......
            ......
            END < label >
```

The statements within the DO loop are obeyed until either the *list* is exhausted or the *logical expression* is no longer true. The *list* control or the WHILE control may appear on their own. If both are omitted the loop is obeyed once. The *list* must be of the same type as the control scalar. The use of a name scalar and list is a particularly useful facility, as illustrated in Section 3.1. DO loops may be nested in the usual way.

The IF statement is an extended form of that available in many languages; it recognises three values of the *logical expression* - true, false and missing. The form is:

IF < *logical expression* > THEN < *cs* > ELSE < *cs* > MISSING < *cs* >

where *cs* represents a single statement or a compound statement. The single statement may not be DO, BEGIN, END or IF.

A compound statement is written as a DO loop. It may have a list control, a WHILE control, both or neither. A second form of IF tests specifically for missing values:

IF < *logical expression* > IS MISSING THEN < *cs* > ELSE < *cs* >

In both forms of IF statement the THEN, ELSE or MISSING phrases may be omitted although at least one phrase must be present.

Two unconditional branching statements are available. These are:

GO TO < *label* >
JUMP < *name scalar* >

The *name scalar* contains the *label* of the statement to which the branch is to be made.

Statements similar in form to their PL/I equivalents are available for input and output.

It is readily seen therefore that ACL has most of the features of a language such as Fortran. However, it is stressed that ACL is not intended as a competitor to Fortran. It has been developed to form the foundation for a number of application systems and provides all the facilities of Fortran for direct use in conjunction with the facilities of these. ACL, being an interpreter rather than a compiler, will run both interactively and in batch. Some additional facilities for program editing and display of values are available to the interactive user. Further statements are included to facilitate communications with application systems as described in Section 4.2.

4.2 The Data Analysis System

The 2900 Data Analysis System (DAS) has a 2-tier structure with the Applications Control Language (ACL) as the first tier and various applications represented in the second tier. The first tier is often referred to as the system level or more specifically as ACL. The second tier is referred to as the subsystem level with each application represented by a subsystem. For example, there is a statistics subsystem which combines with ACL to provide the fullest possible facilities for statistics. Each subsystem includes data management operations for handling the data structures recognised specifically by that subsystem and achieving communication with the general data structures recognised at the system level by ACL. Thus the structure of the Data Analysis System is as shown in Fig. 1, which shows five subsystems.

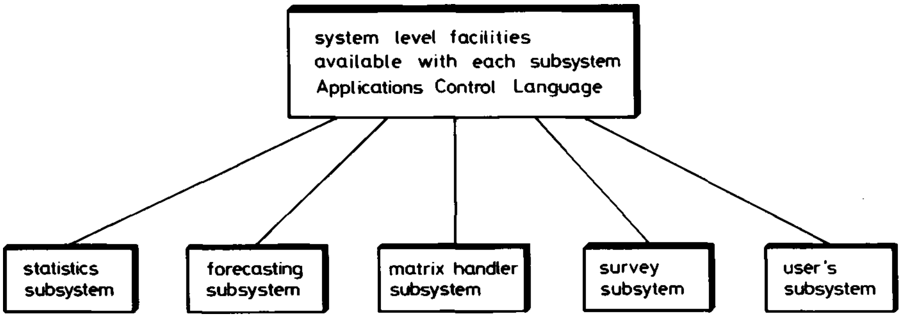


Fig. 1 Structure of the Data Analysis System

At the time of writing the statistics and matrix handler subsystems are available from ICL. Work is in progress on the detailed design for the survey subsystem and an earlier design for the forecasting subsystem is being reviewed. The fifth subsystem shown as a user's subsystem is included to illustrate the fact that further subsystems can be added to meet particular needs.

DAS will run with ACL and all currently available subsystems organised as one complete system. The user may reference any or all subsystems within one program.

Alternatively, each subsystem is available separately with ACL. The statistics subsystem and ACL together are known as 2900 statistics[4] and the matrix handler subsystem and ACL together are known as 2900 Matrix Handler.[5] The 2900 Linear Programming system[6] also uses ACL as its basis and, therefore, has the same structure as 2900 Statistics. Although the LP system has never been considered to be part of DAS it has the full benefit of ACL in the same way as any of the subsystems that are part of DAS.

When included as part of DAS, ACL recognises an INVOKE statement which selects a particular subsystem. The user may then mix statements from the control language of that subsystem with those from ACL. An example of such a mixture is given in Section 3.1, where a REGRESSION statement from the statistics subsystem control language is included within a DO loop which is processed by ACL. Thus a user's program may have, for example, the following structure:

```
BEGIN PROGRAM
 -    ACL statements
INVOKE STATISTICS
 -    STATISTICS and ACL statements
INVOKE FORECASTING
 -    FORECASTING and ACL statements
INVOKE STATISTICS
 -    STATISTICS and ACL statements
END PROGRAM
FINISH
```

This program begins with some ACL statements before calling in the statistics subsystem. Later the forecasting subsystem is called to replace the statistics subsystem which is later recalled. Thus subsystems may be called and recalled as required. At each stage statements from the current subsystem may be mixed with those of ACL. ACL therefore not only provides common facilities to all subsystems but it provides one means of communicating data between subsystems. This is achieved using the facilities for including ACL structures in subsystem statements and the feedback facilities whereby values computed by subsystem analyses are transferred to ACL structures along the lines described in Section 2.3.

5 Subsystems of the data analysis system

Before discussing the different facilities of the individual subsystems it is important to emphasise the common features. The DAS control language is made up of ACL and the control languages for each of the subsystems. Although subsystems have their own control languages designed to meet the requirements of their particular subject areas, they employ common constructs to ensure that they, together with ACL, present a unified appearance to the user. Statements in all subsystem languages have a form which is close to English. They are all based on the same keyword approach and the same routines are used in their interpretation. Three examples from three different subsystems illustrate the common English form:

```
REFORM DATA MATRIX DMEXAM ADDING VARIATES NEW1 AND
   NEW2 AND DELETING VARIATE OLD
INVESTIGATE SERIES S1 USING EXPONENTIAL SMOOTHING
   MODEL LINEAR GROWTH AND PARAMETER 0.3.
INVERT MATRIX A USING CHOLESKI METHOD STORING
   RESULT IN B, RANK IN R AND DETERMINANT IN DA
```

The statements are shown in their full form including a number of redundant words; these emphasise the English nature and serve to aid the user's memory. In addition there are a number of alternatives for certain keywords, particularly to allow abbreviations. Replacement or supplement of keywords by their equivalent in other languages is possible so that the control language could be adapted for use in countries where English is not the native language. Countries with more than one language in regular use could also be catered for.

A second common feature of the subsystem control languages is the inclusion of references to system structures, namely, scalars, arrays and lists. For example, a number of keywords introduce a single value. In all subsystems it is usually possible to present the name of a scalar of appropriate type instead of the value, thus allowing parametric presentation of values. Many statements require specification of lists of various types. The rules for presentation of these lists apply throughout the system. Numeric lists may be presented in a variety of shorthand forms. The list 1,2,3,4,6,8,10 may be written in any of the following ways:

```
1,2,3,4,6,8,10
1:4,6,8,10
1,2,3,4 (2) 10
1:4 (2) 10
```

Alternatively, if AL is the name of a list with values 1,2,3,4,6,8,10, the name AL may be presented instead. Further, if values 11,12 and 13 are to be added to this list for a particular statement, this could be presented as

```
AL '11:13
```

There is a correspondingly wide variety of shorthand ways of presenting lists of names that may be used in all subsystem control languages.

5.1 Statistics subsystems

The statistics subsystem[3] provides a wide range of data-management operations to enable the user to make the best use of what is probably the most comprehensive range of analyses and displays provided so far in any statistics system. Two additional data structures, the *data matrix* and the *data set*, are recognised by this subsystem. They are defined in Sections 5.1.1 and 5.1.2, respectively.

5.1.1 *Data Matrix*

The data matrix, as in most statistical systems, is a rectangular organisation of data values referred to by a single name. The variates (columns), which may be the same types as scalars, namely integer, logical, name or real, are identified by a unique name. The points or cases (rows) are identified by number within the sequence of cases. The subsystem may be aware of any number of data matrices or data sets but only one is 'current' at any one moment. Data analysis statements relate always and only to this current structure.

Access to data values in a data matrix is permitted within the formal organisation of a scan of the cases. In the example which follows the cases have been given the name MEN by the user. The scan begins with a SCAN statement and ends with an END statement, the name MEN being used in these statements to delimit clearly the scan. The operations specified are to set the value of variate X1 and to set variate Y to zero if its value was previously missing. These operations are performed for the first case in the data matrix, then for the second and so on until all cases have been processed:

```
SCAN MEN
X1= X2 + X3/ (X7-X8)
IF Y IS MISSING THEN Y=O
END MEN
```

Statements available in such scans are identical to those in ACL but reference to variates is permitted. The opening of a scan may be considered to be opening a particular kind of block providing access to variates. Thus the scope of variates is confined to such blocks and the scan construct therefore fits with the block structure of the DAS language.

5.1.2 *Data set*

The data set is an extension of the data matrix which it includes as a special case. The terms *cases* and *variates* are used as in the data matrix. In a data matrix the case contains one value for each variate and is of fixed length. In the data set the case consists of a number of segments following a hierarchical structure. Each segment contains one value for each variate and is itself of fixed length. However, segments may appear more than once within a case.

Consider data collected for each of a number of families. A first series of questions is answered once for each family and a second series of questions is answered once for each child in the family. Thus in a data set representing such data a case corresponds to a family and it consists of two types of segments, one containing family details and the other containing child details. One case, therefore, consists of one family segment and a number, possibly zero, of child segments. This example illustrates the simplest hierarchy consisting of two segment types. The hierarchy associated with a data set may contain any number of segments arranged in any branching pattern.

The hierarchy is specified using a TREE phrase consisting of the word TREE followed by segment names separated by one or more uses of the special characters < and > which have the following meaning:

< means go down one level in the hierarchy
> means go up one level in the hierarchy

Use of these characters therefore follows the normal bracketing rules. Consider the following three examples:

```
TREE  FAMILY < CHILDREN >
TREE  FAMILY < CHILDREN < ILLNESS >>
TREE  FARMS < SECTIONS  < CROPS >>< BUILDINGS < ROOMS >>
```

The first describes the two segment hierarchy referred to above while the second extends this to a three segment hierarchy. The third has five segment types arranged as shown in Fig. 2.

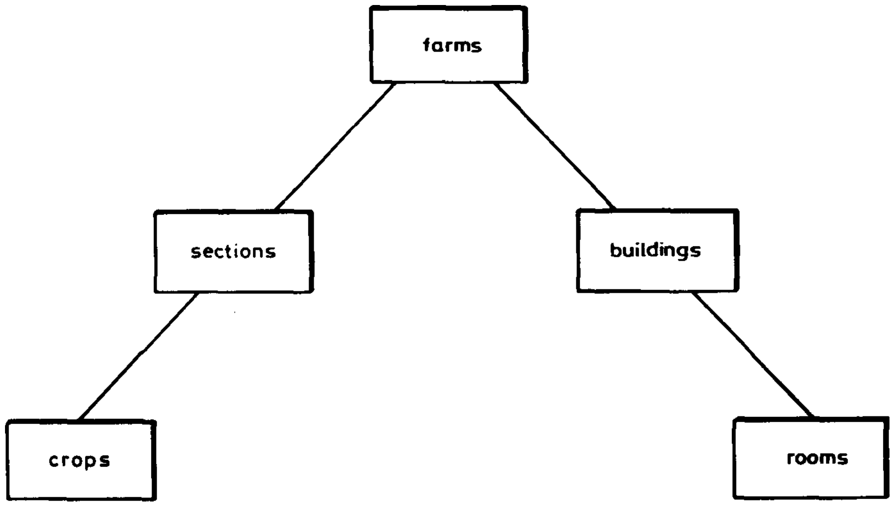


Fig. 2 Data set with five segment types

The scan of cases in a data matrix described has a natural extension to permit access to values in a data set. The scan of a data set may again be considered as following block structure concepts. The scan consists of a nested sequence of scans of segments with the nesting following the hierarchical structure. For example the scan of a data set representing farm data with the hierarchical structure given above would be organised as follows:

```
SCAN FARMS
  -   operations on farm variates
```

```
SCAN SECTIONS
  -  operations on farm and section variates
SCAN CROPS
  -  operations on farm, section, and crop variates
END CROPS
  -  operations on farm and section variates
END SECTIONS
  -  operations on farm variates
SCAN BUILDINGS
  -  operations on farm and building variates
SCAN ROOMS
  -  operations on farm, building and room variates
END ROOMS
  -  operations on farm and building variates
END BUILDINGS
  -  operations on farm variates
END FARMS
```

Thus we see that the rules for access to variates agree with the basic principles of block structures. If the scan of a segment is open then access to variates in that segment is permitted; and the ACL structures of scalars, arrays and lists may be referenced in statements included within scans.

5.1.3 Data management and analysis

A complete description of the many displays, plots, histograms, tabulations and formal analyses is not possible here. An impression of the comprehensive range of analysis facilities available may be gained from the following lists:

Tabulations	-	any number of dimensions
Histograms	-	with distribution fit
Plots	-	scatter diagram, bar charts, probability plots
Summaries	-	whole matrix or in groups
Analysis of variance	-	with polynomial partitioning
Regression	-	multiple, polynomial, stepwise, Beale-Kendall-Mann, Garside 2^n, Element analysis, ridge regression
Components analysis	-	on correlation or covariance matrix
Factor analysis	-	3 different methods, many methods of rotation
Discriminant analysis	-	2 different methods
Canonical analysis		
Cluster analysis	-	3 different methods

Facilities are included in all these analyses for the feedback, either to new variates in the data matrix or to ACL arrays and scalars, of values computed in previous analyses. Thus further analyses may be specified on previous results and sequences of analyses are possible. The stage-by-stage process of data analysis is thus fully recognised.

Analysis and feedback operations are supported by a comprehensive range of data management operations including the creation of derived variates in a data matrix or data set, reading additional data, creating new data matrices or data sets by amalgamation of existing structures or by selecting cases or segments. Data matrices and data sets may be saved on system SAVE files and retrieved later in the same run or in subsequent runs. They may be read in from, or output to, a variety of file types. The user thus has complete flexibility in handling, reorganising and analysing his data.

5.2 *Matrix handler subsystem*

This subsystem recognises vectors and matrices of various different forms and with different properties. Facilities are provided for the input and output of matrices and vectors and for the usual matrix operations including addition, subtraction, multiplication, division, inversion and for computation of eigen values and vectors. English-like statements such as those exemplified in Section 5 are available to select these operations. Alternatively they may be selected using statements of the matrix arithmetic form which may include ACL structures, namely arrays and scalars. A matrix may be declared as an array and hence made available to ACL and to other subsystems; and, conversely, a matrix may be picked up from an array.

Facilities have been designed for implementation in a later release whereby the correlation matrix or its equivalent may be passed between the matrix handler and statistical subsystems. These will allow useful combinations of the facilities of the two subsystems.

5.3 *Forecasting and econometrics subsystem*

A first draft of this system has been completed and is being revised in the light of later developments and appreciation of the needs. Although details may change, the main features are expected to be as follows.

The data structure recognised by the forecasting subsystem is the *series*. This is held in the same form as an ACL array so that ACL and the other subsystems may access series in the same way as arrays. In particular, some of the facilities of the statistics subsystem may be applied directly. The subsystem provides, through its English-based control language, facilities for the input, output, manipulation and analysis of series. The input facilities follow closely the corresponding facilities of the statistics subsystem. Other facilities include:

(*a*) the trial fit of any of a number of models to the whole or part of a series and the production of forecasts

(*b*) the establishment of a particular model with a particular series. Details of the model are retained and are saved with the series if this is saved

(*c*) the production of forecasts for series using their established models (*d*) the updating of series and the comparison between previous forecasts and actual data now supplied.

As with other subsystems, a full complement of feedback and data management operations is provided.

5.4 Survey subsystem

The design of the survey subsystem will be based on the results of a detailed investigation of user requirements which is in progress. In the absence of this subsystem facilities will be provided in the statistical subsystem.

The subsystem will recognise the same data structures, namely the data matrix and the data set, as the statistics sübsystem. There will be complete interchange of these structures between the two subsystems. In fact a data matrix established as the current data matrix by one subsystem will remain current when the other is invoked. Facilities will include input of data, data validation and correction, the formation of frequency and other tables and the display of tables in a wide range of arrangements.

The subsystem will also recognise the *table* as a data structure. This will enable the user to specify operations on tables such as combining tables and performing arithmetic on and comparisons between tables. The saving and fetching of tables will allow the user to retain tables from one run to another. Transfer between a table and an array will permit useful communication with ACL and other subsystems.

6 Database management

Previous Sections have stressed the need for a comprehensive range of facilities for the management of data, to enable the user of the various analysis sytems to make full use of the facilities which they offer. The data-management facilities included in such systems as Package-X and DAS form major parts of these systems and provide a major part of their flexibility in practical use. The next stage is to consider the database and possible relationships between it and these systems. The following Sections deal with the statistical system in this context; similar considerations apply to relations with other application systems.

6.1 A statistics system as database manager

Package-X and the Data Analysis System have facilities for saving the various data structures on system SAVE files for later retrieval. In addition, the user may ask the system to report on the structures currently stored on SAVE files. Thus both systems are able to manage a user's database made up of data matrices, arrays, scalars and lists. DAS can, in addition, include other structures such as the data set. DAS and Package-X can therefore accept the role of manager of a database that would not justify the use of a more comprehensive system such as IDMS.

IDMS[5], available on the 1900 and 2900 series, is a powerful and flexible tool for the management of a database which occupies a central position in an organisation's operation. Such a database requires this power and flexibility and DAS or Package-X could not provide it. However, these systems could provide a useful supplement to the facilities of IDMS. The IDMS system would be responsible for the day-to-day management and with appropriate links between the database and DAS and Package-X, to be discussed in the next Section, these systems could be used to perform any necessary re-arrangement or transformation of the data in preparation for analysis. They could also be used to save data temporarily, away from the database. The statistics system would be managing a separate *ad hoc* database for the duration of the current analysis and thus providing valuable supplementary facilities.

6.2 Links with the IDMS database

The most easily established links between an IDMS database and an application system such as DAS or Package-X are indirect links at file level; i.e. programs using IDMS are written to access the required data and write it to a file in a form which Package-X or DAS can read. Such communications programs will often not be difficult to write or to use. They can provide satisfactory links for those users who are able to define clearly, at the beginning of a study, the data they require to access. If it is likely that the user will wish to change his data access requirements during the analysis, then the indirect links will be noticeably less convenient than more direct links. The user will have to break off his use of the statistics systems, use an appropriate link program to extract the required data and write it to a file, describe the data to the statistics system and initiate its input, before he is able to continue the analysis. This is a purely computing process that the statistician is likely to find an irritating distraction.

It is clear that if the statistics system is capable of accessing the database directly the access process will no longer be a distraction and will fit more comfortably into the analysis process. Thus the user will be able to access data simply as required during analysis. The direct link preserves the interactive analysis approach favoured by most data analysts.

To achieve the direct links the data structures recognised by the statistics system must have a clear and unambiguous correspondence with structures in the database. The data matrix recognised by the statistics system is represented in an IDMS database by a *set*. The scan of a data matrix as exemplified in Section 5.1.1. would correspond to accessing each member record in the set in their linked sequence within the set. The name of the data matrix would normally correspond to the name of the member record type. Thus the statement SCAN FARMS would have a clear and direct interpretation in IDMS terms.

The relationship of the DAS data set containing hierarchically structured data to an IDMS database is particularly interesting. The record types contained in an IDMS database may be linked to form a complex network. The DAS data set corresponds to a number of record types linked in a hierarchical arrangement. For example, the scan of the data set exemplified in Section 5.1.2 involves five record types. This scan pattern again has a clear and direct interpretation in IDMS terms. Thus any subset of record types linked in hierarchical structured relationships may be scanned in this way.

The structure of an IDMS database is defined in the *schema*. Record types and their relationships are part of this definition. Thus, if a statistics system such as DAS or Package-X were to be linked to an IDMS database, it would be able to identify references to record types such as data matrices or data sets by appropriate reference to the IDMS schema. Definition of the data would therefore be permanently available to the statistics systems and it would no longer be necessary to define a matrix or data set to the system before it could be analysed. The system would simply check by reference to the schema that references to record types follow proper relationships appropriate to the data set.

A new system, providing direct access to an IDMS database and bringing together many of the features of DAS, Package-X and IDMS, is being implemented for one particular ICL user. When complete this system should demonstrate the substantial benefits to be gained from the direct links and should show that similar links for DAS and Package-X would be as valuable and should not be difficult to achieve. It is hoped that a description of this system and an account of the experience of using it can be given in a later paper.

References

1 COOPER, B.E.: 'Advances in statistical system design'. *J. Roy. Stat. Soc. A.*, 1977, **140.**
2 Publications on Package-X are in course of preparation. Information is available from Dataskil Ltd. or the Central Statistical Office. Package-X is a Crown Copyright product
3 2900 Statistics Reference Manual: ICL Publication TP 6873
4 IDMS Technical Overview: ICL Publication P 1124
5 2900 Matrix Handler: ICL Publication TP6907.
6 LP2900 : ICL Publication TP6880

Structured programming techniques in interrupt-driven routines

P.F.Palmer
Product Development Group (Southern Development Division), Bracknell

Abstract

The application of structured programming techniques to the production of interrupt-driven code is illustrated by the design of a microcode module, recently implemented on the 2903/4. The Jackson method used provides a powerful means of developing an accurate design and well structured microcode

1 Introduction

Structured programming techniques are popular in some areas of software development, less so in others. These techniques are often analysed in the context of high-level languages and 'commercial' software. However, much system software has characteristics quite unlike the software used as examples of structured programming. Although there is always a trend towards more structured designs, many of the ideas of structured programming have still to find a place here. One area where formal structured programming techniques are little used is the area often called 'firmware' – the low-level system software, including the microcode of machines like the 2903/4. One aspect of this code is that it is frequently interrup-driven; and as will be shown in Section 3, interrupt-driven code is inherently difficult to structure cleanly.

In May 1978 the 2903 microcode team started on the development of a new microcode module to drive a communications coupler, called an SMLCC, a new addition to the set of 2903/4 peripherals. In an effort to improve still further our design techniques we chose to experiment with a design using Jackson structured programming techniques.[1] This paper is the story of how the design went and what was achieved. The result of the experiment was a success, rather more so than had been expected at the outset. As well as producing a self-documenting design and well organised code it elucidated several features of the interrupt-driven code.

The Jackson techniques turned out to be almost purpose-designed for our application. The paper presents the design of the new microcode module from the initial stages of writing down the data structures representing the way the microcode views the transmission blocks on the line, to the point of writing the microcode. The design technique throws light on several aspects of the code, and some of the features which come out of the design are clearly relevant to any interrupt-driven package. Comparison with other 2903/4 microcode shows that the code produced is at least as good as other implementations. With the benefit of such a clear design methodology the approach must be strongly recommended.

2 The problem

In this Section we outline the way in which the microcode module fits into the 2903/4 system. Of necessity some understanding of programming terms is required; a glossary is given in Appendix 1.

The microcode we are going to discuss drives an SMLCC coupler which is connected to a communications line on which there may be several terminals, say 7181 videos or 7502 remote job entry terminals. The line protocol can be ICLC01, ICLC02, or ICLC03, with the 2903/4 acting as the primary; for a definition of these protocols see, for example, Reference 2. The coupler is operated in a character-by-character manner, interrupting the microcode when it requires the next character. An interrupt mechanism is a fundamental part of all modern computers and it is not the purpose of this paper to describe it. However, to set the context for which our design is intended we shall describe the operation in a little more detail.

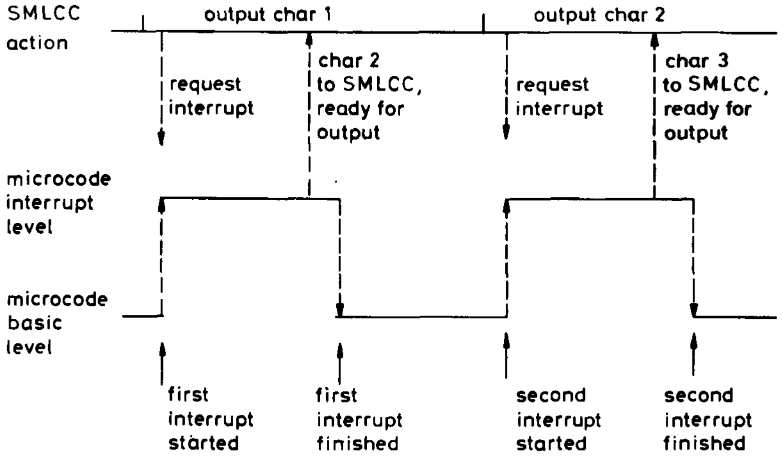


Fig. 1 Timing diagram

When the 2903/4 executive wishes to communicate with a terminal it passes the addresses of two buffers to the microcode with a special START instruction. The first buffer contains data to be transmitted to the line, for example a status poll. The second is a buffer for the response of the terminal, say a status response. The microcode's function is to move data from the output buffer to the SMLCC until it recognises a character which terminates the output. This character is transmitted, followed by a block-check character (BCC) if required. The microcode then stops transmitting, puts the line into receiving mode and waits for a response. Incoming data are transferred from the coupler to the store until again a terminating character is detected, when the microcode desynchronises the line.

The SMLCC handles one ISO line character at a time. On output, when it has serialised a character onto the line, it interrupts the microcode, requesting another character. The microcode fetches the next character from the store, passes it to the coupler and exits from the interrupt. This sequence continues until the end of the transmission block. Reading data works in a similar way.

When the SMLCC requests an interrupt the hardware of the 2903/4 stops the execution of the system in its basic level and automatically switches the microcode into its interrupt level which then starts executing from the interrupt entry point. The microcode runs on in interrupt level until the character from the SMLCC has been processed and then obeys a special INTERRUPT EXIT instruction which causes the hardware to restart the basic level at the point at which it was when the interrupt occurred. A timing diagram for this process is given in Fig. 1.

As can be seen the interrupt servicing microcode is executed in a series of paths, each one separated from the next by an interval spent executing a quite different piece of microcode. The design problem that is specific to such interrupt-driven code is to decide how to remember what happened between one interrupt and the next.

3 The microcode design

3.1 *Method*

The design method adopted was based on some of the ideas expounded by M.A.Jackson.[1] The basic technique is to write down the design first as a simple structured diagram and then as pseudocode. The latter, however, is written not as if it were interrupt-driven but as if the data were always available and could just be written or read as a serial file, using READ or TRANSMIT routines. At this level the fact that the code is interrupt-driven is completely disguised and as a result the structure is clear and easy to follow. The program is then 'inverted' to produce pseudocode which is interrupt-driven but has a less obvious structure. This 'inversion' is a purely mechanical process and may be viewed as just an implementation of the design; it is explained in more detail in Section 3.3. The term is due to Jackson; it does not have the same meaning as, for example, inversion of an indexed file. After the inversion, additional features for handling interrupts, such as timers, are added.

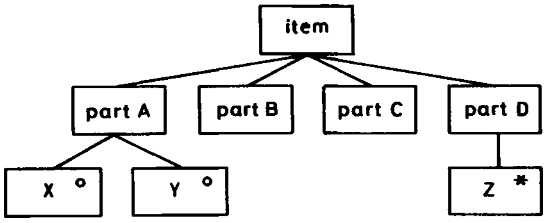


Fig. 2 Tree diagram

3.2 Initial design

First we have to describe the transmission blocks in more detail. The notation used by Jackson[1] is suitable for this; to summarise, it uses a tree diagram for the data, as in Fig. 2.

Fig. 2 shows 'item' made up of a *sequence* part A, part B, part C, part D. A small circle indicates a *selection*, so part A is shown as made up of X *or* Y. A box with an asterisk means that the item may be repeated an unspecified number of times; thus part D is Z or ZZ or ZZZ etc. 'Item' could therefore be, for example, X part B part C ZZ or Y part B part C ZZZ. In this notation the microcode's-eye view of the transmission is given in Figs. 3-5. Terms such as SYN, ETB are defined in the Glossary, Appendix 1.

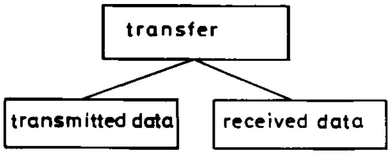


Fig. 3 Data tree

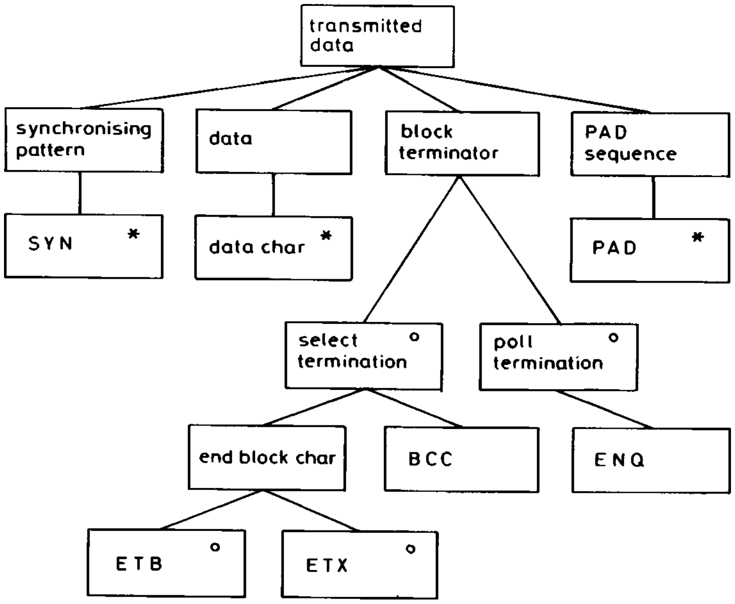


Fig. 4

To the microcode, there is no significance in the different types of transmission block, say between a poll or a select, although they will be significant to higher levels of software. A full definition of the data may be found in several places, for example, Reference 2.

Note however that the microcode does make some distinction between text and status responses. On a text response, the whole block is translated to 2903/4 3-shift code before it is placed in the buffer. On a status response, it is not. Although the reason is minor, it does affect the design.

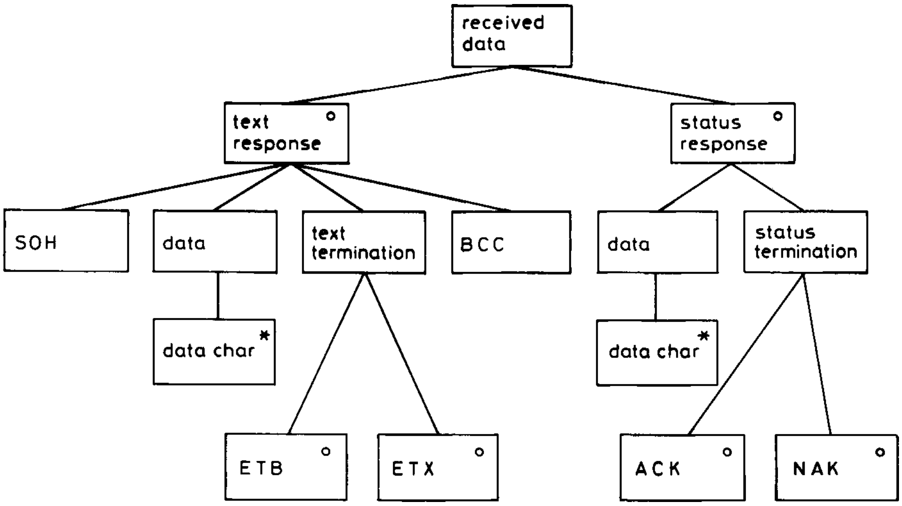


Fig. 5

Once we have the data structure, again following Jackson, we draw a tree diagram for the code based on the data structure. We did not use the code tree very much in our original design, since it is relatively easy to write down the pseudocode directly from the data diagram. For a problem where the actions to be performed are more intricate, a code tree is a useful tool. For completeness a simplified code diagram is given in Figs. 6-8. Notice how the structures match those of Figs. 3-5.

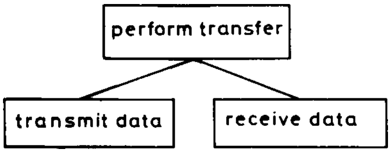


Fig. 6 Simplified code tree

The next step is to represent the design in more detail as pseudocode. For our problem, this step is straightforward, and the code is given in Fig. 9. Notice again how the structures of Figs. 6-8 and Fig. 9 match. Boxes with ° map onto conditional statements (*if*..*fi*), boxes with an asterisk map onto loops (*until do repeat*). For the first time, we can see how the choice on selections is made, and how loops terminate.

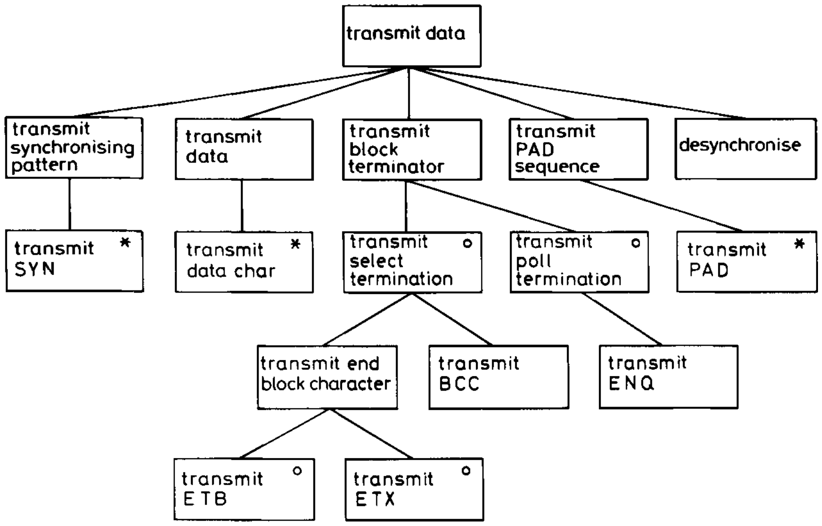


Fig. 7 Simplified code tree

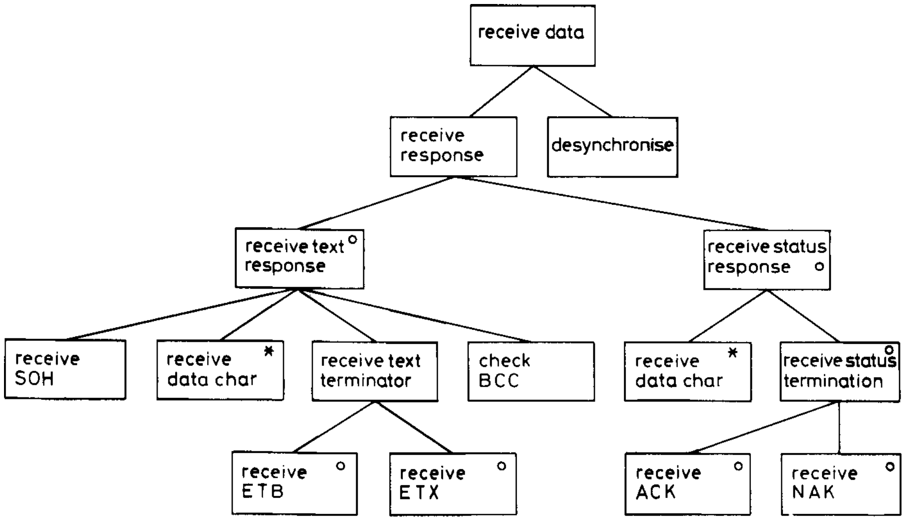


Fig. 8 Simplified code tree

The 'transmit' and 'read' calls in Fig. 9 are the way data are output to, and taken from the line. Representing the input and output in this way is a major feature of the design method. We know the final microcode cannot look like this, since it is interrupt-driven. However, pretending it can makes the pseudocode clear and easy to follow.

```
          set SYN count = 2;
          c In the simple case, two SYN chars are sent
          until     SYN count = 0
          do        decrement SYN count by 1;
                    transmit (SYN)                               c{SEND SYN}
          repeat;
          c Two SYN chars have been output. Now the data can be sent.
          until     fetch next data character from store;
                    data character = ETB or ETX or ENQ
          do        transmit (data character)                    c{SEND CHAR}
          repeat;
LABEL A:  if        data character = ETB or ETX
          then      transmit (ETB or ETX);                       c{SEND BCC}
                    transmit (BCC)                               c{SEND PAD1}
                    c The SMLCC calculates the BCC automatically
          else      c Assume ENQ
                    transmit (ENQ)                               c{SEND PAD2}
          fi;
          set PAD count = 2;
          c We put two PAD characters at the end of the output
          until     PAD count = 0
          do        decrement PAD count by 1;
                    transmit (PAD)                               c{SEND PAD3}
          repeat;
LABEL B:  desynchronise line;
          c That is the output complete, now read the reply.
          read (char);                                   c{READ FIRST CHAR}
          if        char = SOH
          then      until   char = ETB or ETX
                    do      convert char to 3-shift;
                            put char in buffer;
                            read (char)     c {READ CHAR AND CONVERT}
                    repeat;
                    convert char to 3-shift;
                    put char in buffer;
                    read (BCC);                                  c{READ BCC}
                    set error status if BCC incorrect
                    c Again it is the SMLCC which does the BCC calculation.
          else      c Assume status response
                    c For the convenience of the system software, the status
                    c characters are not translated to 3-shift.
                    until   char = ACK or NAK
                    do      put char in buffer;
                    read (char)                                  c{READ CHAR}
                    repeat;
                    convert ACK or NAK to 3-shift;
                    put char in buffer
          fi;
          desynchronise the line;
          c The transfer is complete.
```

Fig. 9 Structural pseudocode

3.3 *Inversion about the read/transmit routines*

Fig. 9 shows the design so far. In some ways, it can be regarded as the complete design, since all the essential flow is there. For example, if it became necessary to modify the design to terminate a status response on an additional character, then the place to change Fig. 9 is clear. However, it assumes data are available by means of subroutines when it is required; an assumption that is incorrect because it is the SMLCC that decides when a character is wanted, and interrupts.

To obtain the interrupt-driven code we 'invert' Fig. 5 about the read/transmit routines. This inversion is performed below. Its great importance is that it is a mechanical process which can be performed in a systematic way on the pseudocode.

We achieve the inversion by a systematic substitution of code. First, we pick out and label uniquely each read or transmit call. In Fig. 9 the label is enclosed in { } as a comment. Now we pick one word of store which we call a *state variable*. We use this state variable as a row of flags. To each of the labelled calls we picked out earlier, we associate a flag. Now, we take, for example,

transmit (ENQ) *c* {SENDPAD2}. *c*

For this call, we substitute the code in Fig. 10.

```
            give character to SMLCC;
            set 'sendpad2' flag in state variable;
            interrupt exit;

SENDPAD2:   clear 'sendpad2' flag.
```

Fig. 10

In addition, at the interrupt entry point we add the code

if 'sendpad2' flag set *then goto* SENDPAD2 *fi*

Notice that we have used the convention writing the labels introducing the inversion in italics to make them stand out.

We make this substitution at all the marked calls in Fig. 9. The code is expanded somewhat, so in Fig. 11 we show only the expansion of the code between LABEL A and LABEL B.

In Fig. 11, we can see for the first time how the individual interrupts are routed. Notice how the interrupts cut right across the block structure; an interrupt will start by jumping into the middle of one block and finishing in the middle of another. The diagram illustrates vividly why interrupt-driven code is difficult to structure.

```
if data character = ETB or ETX
then                      give char to SMLCC;
                          set 'send BCC' flag in state variable;
                          interrupt exit;
SEND BCC:                 clear 'send BCC' flag;
                          c This completes the first transmit.
                          c Now start the next transmit.
                          tell SMLCC to send BCC;
                          set 'send pad1' flag in state variable;
                          interrupt exit;
SEND PAD1:                clear 'send pad1' flag;
                          c This completes the second transmit.
else c Assume ENQ.
                          give char to SMLCC;
                          set 'send pad2' flag in state variable;
                          interrupt exit;
SEND PAD2:                clear 'send pad2' flag;
fi;
set pad count = 2;
until pad count = 0
do                        decrement PAD count by 1;
                          tell SMLCC to send PAD;
                          set 'send pad3' flag in state variable;
                          interrupt exit;
SEND PAD3:                clear 'send pad3' flag
repeat;
c That is the end of that code.
c At the interrupt entry point we will have the code.
INTERRUPT ENTRY:          if 'send BCC' flag set then goto SEND BCC  fi;
                          if 'send pad1' flag set then goto SEND PAD1 fi;
                          if 'send pad2' flag set then goto SEND PAD2 fi;
                          if 'send pad3' flag set then goto SEND PAD3 fi;
```

Fig. 11 Inverted pseudocode

The purpose of our state variable is to remember from one interrupt to the next where one finished and where the next should start. A state variable is a standard feature of any interrupt-driven code, although in many examples of such code, it is not always clear quite what constitutes the state variable.

The flag word we used is only one way of implementing a state variable. The state variable could have been a value to be used as an index to a jump table, or could have been the microprogram address of the return point. This is an implementation decision. We chose this implementation on 2903/4 because there are some excellent conditional-jump-on-bit instructions, but implementing a jump table is costly.

3.4 *Completion of pseudocode*

The pseudocode is only two stages away from completion. First we implement all the *do repeats* and *if then fi* with simple *if then goto fi* statements. Once this is finished, looking at Fig. 12 we can see some obvious optimisations to make. For example, having returned to label SENDPAD1 on interrupt entry, the code jumps smartly to SENDPAD. The optimisation is to jump directly to SENDPAD on interrupt entry.

```
                     if data character = ETB or ETX
                     then goto TRANSMIT ENQ fi;
                     give char to SMLCC;
                     set 'send BCC' flag in state variable;
                     interrupt exit;
        SEND BCC:    clear 'send BICC' flag;
                     tell SMLCC to send BCC;
                     set 'send PAD1' flag in state variable;
                     interrupt exit;
       SEND PAD1:    clear 'send PAD1' flag
                     goto TRANSMIT PAD;
    TRANSMIT ENQ:    c Assume ENQ.
                     give char to SMLCC;
                     set 'send PAD2' flag in state variable;
                     interrupt exit;
       SEND PAD2:    clear 'send PAD2' flag;
    TRANSMIT PAD:    set PAD count = 2;
        SEND PAD:    if PAD count = 0 then goto END OF OUTPUT fi;
                     decrement PAD count by 1;
                     tell SMLCC to send PAD;
                     set 'send PAD3' flag in state variable;
                     interrupt exit;
       SEND PAD3:    clear 'send PAD3' flag;
                     goto SEND PAD;
END OF OUTPUT:
c That is the end of that code.
c At the interrupt entry point we will have the code.
 INTERRUPT ENTRY:    if 'send BCC'  flag set then goto SEND BCC  fi;
                     if 'send PAD1' flag set then goto SEND PAD1 fi;
                     if 'send PAD2' flag set then goto SEND PAD2 fi;
                     if 'send PAD3' flag set then goto SEND PAD3 fi;
```

Fig. 12 Pseudocode before optimisation

To finish the design we add one more feature. The microcode must be resilient to an expected interrupt not appearing, perhaps because the modem has failed or the terminal does not respond. Therefore, before each exit, on transmission paths, a timer is started, and is cleared when the next interrupt occurs. On read the whole message is timed out. The timer fail code is not shown. In the microcode module we are designing, if a timer fail occurs, the whole transmission is aborted, and in our design we did not use Jackson techniques for this relatively easy part of the code. However it is easy to see ways to introduce the timer fail case, if it were necessary.

Addition of the timer code completes the pseudocode and our design. Fig. 13 shows the final version of the code in Fig. 11. Appendix 2 contains all the final pseudocode.

```
                  set PAD count = 2;
                  if data character = ETB or ETX
                  then goto TRANSMIT ENQ fi;
                  give char to SMLCC;
                  set 'send BCC' flag in state variable;
                  start timer; interrupt exit;
      SEND BCC:   clear timer; clear 'send BCC' flag;
                  tell SMLCC to send BCC;
                  set 'send PAD' in state variable;
                  start timer; interrupt exit;
  TRANSMIT ENQ:   give char to SMLCC;
                  set 'send PAD' in state variable;
                  start timer; interrupt exit;
      SEND PAD:   clear timer; clear 'send PAD' flag;
                  if PAD count = 0 then goto END OF OUTPUT fi;
                  decrement PAD count by 1;
                  tell SMLCC to send PAD;
                  set 'send PAD' in state variable;
                  start timer; interrupt exit;
END OF OUTPUT:
c That is the end of that code.
c At the interrupt entry point we will have the code.
 INTERRUPT ENTRY: if 'send BCC' flag set then goto SEND BCC fi;
                  if 'send PAD' flag set then goto SEND PAD fi;
```

Fig. 13 Final pseudocode after optimisation and addition of timers

4 Characteristics of the design

The steps to the final design were:

Step 1 produce data structure diagrams
2 produce code structure diagrams
3 write pseudocode based on structure diagram, treating data as if it were from an immediately accessible serial file
4 invert the pseudocode
5 add interrupt specific code.

We took the opportunity to optimise at several of these stages, and examination of the final pseudocode shows no obvious design-induced inefficiencies. In Section 5 we compare the microcode produced from this design with other microcode, and show that it is at least as good when measured in terms of size and pathlengths.

The final pseudocode superficially shows none of the structure visible earlier. The nesting, clear in Fig.9, has all but disappeared. However, should we ever have to modify the code, we will be able to use the diagrams from step 3.

If the final pseudocode is examined, there is still visible a consistent organisation which is inherited from the earlier stages. We suggest these are characteristics which improve the structure of any interrupt driven code.

Characteristic 1

A path through the code is clearly partitioned into the sequence

(*a*) interrupt handling code
(*b*) switch code, from interrupt to inline code
(*c*) inline code
(*d*) interrupt exit.

The code itself is structured into units of independent inline code, with each unit performing only one action. Diagrammatically this can be represented as in Fig.14.

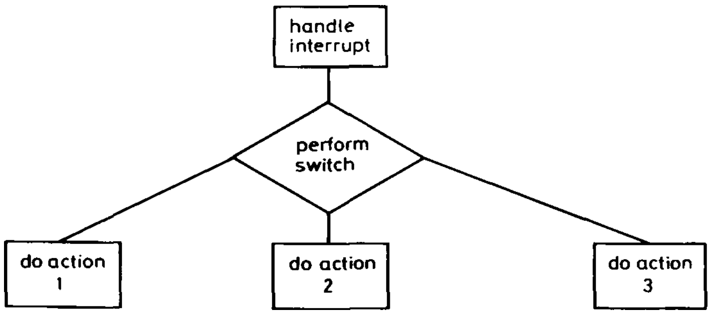


Fig. 14 Structure of code

Characteristic 2

The state variable is handled consistently. There are some implied rules:

Rule 1 state variables are clearly identified and not intermingled with other flags
Rule 2 state variables are accessed once only during each interrupt
Rule 3 state variable for next interrupt is set after in-line code.

In our pseudocode, Rule 2 above was enforced by actually clearing the flag after use. This keeps the flag word tidy. The rule is:

(*a*) interrupt pending - one and only one flag set
(*b*) interrupt being actioned - no flags set.

Rule 3 above is important structurally. Consider the difference between the two pieces of pseudocode in Figs. 15 and 16.

```
set state variable
Action 1
Action 2
exit
```

Fig. 15

```
Action 1
Action 2
set state variable
exit
```

Fig. 16

In execution, there can be no difference. However the code of Fig. 15 is structurally weaker. Once it is implemented, changing action 2 so that it has a choice of exits will have a drastic effect.

There is one further point to be made. In Fig. 9 we used the technique suggested in Reference 1 of 'reading ahead' for the first character of the incoming block. The read ahead is the first read after turning round the line.

Following the read ahead through the inversion shows that the effect is to introduce a special once-entered read-first-character piece of code. In the final version, this code has the special actions of clearing the timer used to detect no response from the terminal and starting the timer on the whole input message. It is instructive to see how this read-ahead rule generates this code in a natural way, and that first-character actions do not have to be inserted in an *ad hoc* fashion.

5 Resulting microcode: assessment and comparisons

The microcode which has its design described in Section 3 represents just under half the final microcode written for the SMLCC. The other half is the code which supports the 2903/4 Executive, makes the line connection, updates timers etc. This has a simple structure and the pseudocode was produced by writing it down directly. We are able to compare the two halves and Table 1 gives the number of errors discovered during testing and validation.

Here, design error means an error where the microcode faithfully represented the intention of the design but the intention was later found to be incorrect. Coding error means an error where the microcode did not faithfully represent this intention.

Table 1

	code designed using techniques of this paper	code designed conventionally
number of design errors	1	6
number of coding errors	15	31

The improvement in design accuracy vindicates the method. The improvement in coding accuracy is also a direct benefit of having a design which has been carefully worked through and which is available in detail. The one design error noted was actually an undocumented hardware restriction which was discovered the hard way on the machine – it had nothing to do with the design method.

We are also in a position to compare our new code with another module written earlier to drive the integrated communications coupler. Direct comparison is difficult: the couplers are different, the line-connection and error-reporting facilities are improved in the newer module. In addition, the way the data are held differs: the earlier module has four dedicated 'working store' locations while the new one has none and consequently requires more instructions to access a given piece of data. The comparison is therefore subjective but is still worth making.

The earlier code has obviously been carefully optimised to reduce the length of the most common path, and the other paths have increased lengths. The new code is more even over all path lengths and there is still room for reducing the lengths of the frequently used paths by jumping on the less frequent cases. When we measure the number of changes of control (jumps) between different parts of the code in one interrupt, we find the two are the same on the standard paths but the new code has fewer jumps on the unusual paths, for example when an ACK or ETB is received. As we showed in Section 4 (Fig. 14), we should expect our design method to give this improvement.

The new code is larger, but this is mainly accounted for by the differences in couplers and facilities mentioned above. Overall, the new code is clear and has an obvious organisation.

Given the fact that some reduction in path lengths is still possible, the comparison shows that the new code is at least as good as the earlier module; and we expect, of course, to have the benefits of easier maintenance from our design.

6 Conclusions

We have used a structured programming technique,[1] to provide a design for an interrupt-driven microcode module for 2903/4.

The strategy is to produce a block-structured design which disguises the interrupts by assuming the communications coupler can be driven as a serial file. This design is then 'inverted' to produce the interrupt-driven code.

The idea of inversion is crucial to the technique. It provides a simple mechanism to identify conceptually the two different types of code organisation, and a procedure for converting from one to the other. The code can be designed in a block-structured way, which gives substantial benefits in having a 'self-documenting' design and well organised code.

The application of this technique to the 2903/4 microcode was very successful, and produced high-quality code which contained almost no design errors.

Structured programming techniques are not widely used in low-level firmware. Our example shows that they can be used with good effect, and that the Jackson technique in particular should always be considered when interrupt-driven code is being designed.

Acknowledgments

A.W.Radgick, Senior Programmer, implemented the SMLCC microcode.

References

1 JACKSON, M.A. *Principles of program design* (Academic Press)
2 TAS 11 ICL. XBM

Appendix 1

Glossary

This paper assumes some understanding of programming terms. This glossary offers additional explanations, especially of the communications terminology.

pseudocode	Code written to represent the microcode. It is not compiled, but used as a template for the final microcode. We choose to write it to look like S3.
flag	A piece of store to remember a true or false condition. The flag can be set (true) or clear (false).
SMLCC	A communications coupler, i.e. a piece of hardware connected to the communications line, which operates the line under the control of the microcode.

ISO codes	One particular way of encoding data onto a communications line. The ISO characters mentioned in this paper are:

SYN	a character transmitted at the start of all blocks to get the two ends of the line in step (synchronisation).
ETB, ETX	two characters used to mark the end of test blocks. As far as the microcode is concerned, the two are interchangeable.
BCC	Block check character, used to detect errors in a block. Small blocks, like status response, do not have a BCC.
ACK, NAK, ENQ	ENQ marks the end of a poll, ACK or NAK marks the end of a status response.
SOH	a character put at the beginning of a text block.
PAD	the character used to round off all blocks.

Appendix 2 Final pseudocode

```
c The code starts initially in the basic level, processing
c the 'START' command from the 2903/4 executive.
                                        set SYN count = 1;
                                        give SYN to SMLCC;
                                        set 'send SYN' flag in state variable;
                                        start timer;
c This first SYN wakes up the SMLCC.
c It will output this SYN and then interrupt for all subsequent
c characters.
•                                       continue processing 2903/4 program;
c The transfer has been started. The basic level goes on to
c do other work.
c All the following code is obeyed in interrupt level.
                          SEND SYN:     clear timer; clear 'send SYN' flag;
                                        if SYN count = 0 then goto SEND CHAR fi;
                                        decrement SYN count by 1;
                                        give SYN to SMLCC;
                                        set 'send SYN' flag in state variable;
                                        start timer; interrupt exit;
                        SEND CHAR:      clear timer; clear 'send char' flag;
                                        fetch next data character from buffer;
                                        if char = ETB or ETX
                                        then goto TRANSMIT ETBX fi;
                                        if char = ENQ then goto TRANSMIT ENQ fi;
                                        c some changes here from fig. 13 for optim–
                                        isation
                                        give char to SMLCC;
                                        start timer; interrupt exit;
                   TRANSMIT ETBX:       give char to SMLCC;
                                        set 'send BCC' flag in state variable;
                          start timer; interrupt exit;
            SEND BCC:     clear timer; clear 'send BCC' flag;
                          tell SMLCC to send BCC;
                          set 'send PAD' flag in state variable;
                          start timer; interrupt exit;
        TRANSMIT ENQ:     give char to SMLCC;
                          set PAD count = 2;
                          set 'send PAD' flag in state variable;
                          start timer; interrupt exit;
            SEND PAD:     clear timer; clear 'send PAD' flag;
                          if PAD count = 0 then goto END OF
                          OUTPUT fi;
                          decrement PAD count by 1;
                          tell SMLCC to send PAD;
                          set 'send PAD' flag in state variable;
                          start timer; interrupt exit;
       END OF OUTPUT:     desynchronise SMLCC;
c That is the whole output block transmitted.
c The SMLCC will automatically synchronise on the incoming message.
c All we have to do is wait for the interrupt.
                          set 'receive first char' flag in state variable;
                          start timer; C Allowing time for the terminal
                          to respond. interrupt exit;
  RECEIVE FIRST CHAR:     clear timer; clear 'receive first char' flag;
                          start timer on whole of input message;
                          c There is time for a 2000 character message.
                          take char from SMLCC;
                          if char = SOH
                          then goto RECEIVE STATUS fi;
                          goto RECEIVE TEXT;
READ CHAR AND CONVERT:    clear 'read char and convert' flag;
                          take char from SMLCC;
        RECEIVE TEXT:     if char = ETB or ETX
                          then goto RECEIVE ETBX fi;
                          convert char to 2903/4 3-shift;
                          put char in buffer;
                          set 'read char and convert' flag in state variable;
                          interrupt exit;
        RECEIVE EXTB      convert char to 2903/4 3-shift;
                          put char in buffer;
                          set 'check BCC' flag in state variable;
                          interrupt exit;
           CHECK BCC:     clear 'check BCC' flag;
                          if BCC incorrect then set error status fi;
                          c The SMLCC does this calculation for us.
                          goto DESYNCHRONISE;
           READ CHAR:     clear 'read char' flag;
                          take char from SMLCC;
```

```
         RECEIVE STATUS:  c This path is taken if the first char of the
                          c incoming block is not SOH, implying a status
                          c response.
                          c The status characters are not translated but
                          c the terminating ACK or NAK is.
                          if char = ACK or NAK
                          then goto RECEIVE ACK NAK fi;
                          put char in buffer;
                          set 'read char' flag in state variable;
                          interrupt exit;
        RECEIVE ACK NAK:  convert char to 2903/4 3-shift;
                          put char in buffer;
         DESYNCHRONISE:   desynchronise SMLCC;
                          clear timer;
c That is the end of the transmission. After tidying, a peripheral interrupt is set to
c the 2903/4 executive. There is then the final interrupt exit.
                          interrupt exit;
c At the interrupt entry point there is the code.
INTERRUPT ENTRY:        if 'send SYN' flag set then goto SEND SYN fi;
                        if 'send char' flag set then goto SEND CHAR fi;
                        if 'send BCC' flag set then goto SEND BCC fi;
                        if 'send PAD' flag set then goto SEND PAD fi;
                        if 'receive first char' flag set
                        then goto RECEIVE FIRST CHAR fi;
                        if 'read char and convert' flag set
                        then goto REACH CHAR AND CONVERT fi;
                        if 'check BCC' flag set then goto CHECK BCC fi;
                        if 'read char' flag set then goto READ CHAR fi;
```

The content addressable file store - CAFS

V.A.J. Maller
ICL Research & Advanced Development Centre,
Fairview Road, Stevenage, Herts.

Abstract

The ICL content addressable file store (CAFS) is an autonomous unit to which a mainframe computer can devolve information retrieval tasks. Very complex selection criteria can be handled and very high rates of search can be achieved, typically one to two orders of magnitude greater than with conventional methods. This paper describes the equipment and the principles on which its operation is based and gives examples of its performance.

1 Introduction

The use of specialised hardware to improve the performance of data management systems is currently providing a central theme to much research, and the literature now abounds with references to such work.[1] Although there is a rich variety of opinions as to the characteristics of such hardware, there is general agreement that there is more potential in this approach than on further improvements in software.

The underlying motive behind these developments arises from a growing realisation that conventional systems, based on the classical von Neumann processing concepts, are unable to meet current and expected user demands, in both performance and facilities, in an effective and efficient manner. The changing pattern of data processing from batch to interactive working has exacerbated the situation, and nowhere is this more apparent than in large real-time systems in which there is significant structural complexity in the stored information.

In the past, levels of performance have been largely determined by a combination of software ingenuity and raw central processor power. Now, however, the technological virtuosity of the semiconductor industry has enabled hardware concepts to be realised cost-effectively that hitherto have been impractical, if not indeed impossible. This new found freedom, when fully exploited, heralds radical changes in much established practice for system design.

The content addressable file store represents one such concept.[2]

2 Rationale for content addressing

Storage devices with content addressable or associative properties have been discussed frequently in the literature for many years. However, whilst their utility has been acknowledged, the technology to provide a device of anything more than trivial size has been lacking. In all cases the objective has been to construct a store which may be accessed directly by using the intrinsic properties of the data items themselves as keys, rather than to rely on some explicit referencing structure.

The concept is therefore not new. Indeed the requirement for such devices arises from the observation that from the earliest days much data processing has been concerned with extracting relevant information from files using single or multiple key matching. This method of file searching is so taken for granted that it is often not appreciated that it is content addressing. In fact, in many installations as much as 60% of available machine time may be used in serial searching of one form or another for such operations as report generation, selective updating and file maintenance.

To execute such tasks on conventional equipment necessitates each record in the target file being brought into the mainframe, tested for relevance and then either discarded or processed in some way to an output file. Such operations are intrinsically inefficient since the real activity on a file in any particular run is determined by the number of external events since the previous run, and this is likely to be low. Indeed in many applications 'hit rates' on files may not exceed 5% of the records present.

During the period when the magnetic tape recorder was the only form of backing storage such inefficiencies were tolerated as if they were part of the natural order, but who, given freedom of choice, would willingly record his information on a piece of material half an inch wide and half a mile long with the only means of access being to reel through it from end to end?

The advent of the moving arm disc file provided an extendable pseudorandom access store and thereby enabled indexing schemes to be developed which, it was hoped, would ensure that only records having a high probability of relevance would be retrieved. The use of such schemes was mandatory for online interactive working where high selectivity of retrieved data and rapid response were required. However, a disc store is not a true random access device and consequently it is only possible to index a file efficiently along one access dimension. Moreover, indexes can possess immense nuisance value. For simple index sequential and random files the overheads in most cases are acceptable. However, as soon as secondary data items are required as keys, the number and average size of the indexes may begin to grow alarmingly as the volume and complexity of the primary data increase. There are, in fact, instances where the indexes can occupy between two and four times the volume occupied by the data to which they refer – a perfectly absurd situation.

Difficulties with indexes are of two kinds. First there is the sheer manipulation required, and secondly there are the complexities of maintenance. Updating of primary data may spawn multiple update tasks which may involve substantial processing and be a very severe overhead in a real time environment. Also, it should not be assumed that indexable operations cover the totality of useful functions. There are, for example, occasions where the relationships between two or more data items within the same record may constitute a selection criterion. Indexes are really very primitive projections of files and consequently their utility should not always be taken for granted.

A further problem of particular relevance in the real time environment is that of how to cope with queries containing imprecisely defined search arguments. Such queries often have the term 'fuzzy matching' applied to them, and indeed this endearing expression arose from a recognition of the human genius for imprecision. An efficient solution to this problem is clearly vital if acceptability of information systems to the lay user is to be achieved.

So far only files have been discussed and not databases. Although there are various views as to what constitutes a database, there is general agreement that the term implies a coherent set of data of greater generality than a simple file which, moreover, can be shared by a wide variety of application programs. This sharing of data raises another problem. Each program may have its own particular logical view of the data which has to be mapped to the stored data. These mappings are often complex due to the impossibility of designing a storage structure which will efficiently satisfy the diverse requirements of different application programs. The consequences of this for conventional implementations are often to interlock the logical and physical structures with complex indexing and linking systems, and thereby make reorganisation difficult and evolutionary growth virtually impossible without complete recompilation. Nevertheless current database techniques permit systems to be built which can satisfy a wide variety of operational requirements for both batch and real time working where these can be accurately predefined.[3] They are much less efficient when it comes to handling *ad hoc* queries, particularly those involving multi-key record selection criteria. Such queries in many cases can only be handled as background batch tasks, and this, although better than nothing, is irritating to many end users who would much prefer to operate interactively.

Although conventional von Neumann processing techniques provide solutions in the functional sense to the problems outlined above they are unable to do so efficiently, and most importantly they restrict interactive working to simple transactions. The system designer is therefore in a dilemma; either he can provide a comprehensive range of selection facilities in a software package which relies heavily on serial search in batch mode, or he can offer an elaborate indexing scheme which at best will give limited real time facilities with only moderate performance, and at considerable cost in terms of index compilation and general updating.

The clue to a possible solution lies in the fact that despite the manifest inefficiency of serial search it is often the only solution and the concomitant penalties have to be grudgingly accepted. It is therefore pertinent to investigate the possibility of building an autonomous searching engine to perform this task and thereby relieve the mainframe of a rather trivial, but nevertheless mill-hogging, comparison and test procedure.

3 An autonomous searching engine

There has been considerable discussion in the literature regarding the possibility of subcontracting the entire data management function to a specialised unit, the data base machine, operating autonomously from the central processor or mainframe. Although this form of functionally decentralised architecture is potentially feasible, many problems remain to be solved if efficient and reliable implementations are to be achieved. Nevertheless substantial performance improvements should be obtainable if certain search and retrieve functions are implemented in the storage subsystem under the control of a mainframe resident data management system.

Several groups have pursued this approach[4-7] and during the last few years a team at the Research and Advanced Development Centre of International Computers Limited has developed a machine known as the Content Addressable File Store (CAFS).[8-11] This is a disc file subsystem containing specialised hardware operating under software control, but using parallel processing techniques for implementing multi-factor selection across either single files or the join of multiple files. The essential requirements placed upon the hardware in this system are those of concurrent execution of powerful selection and retrieval functions on multiple data streams arising from the simultaneous reading of many disc channels. Although these features have been realised in conjunction with the moving arm disc file, they are applicable to other cyclic random access storage devices such as magnetic drums, fixed head discs, bubble memories etc. An important principle in the design was to resist the temptation of using exotic technology by using only 'state-of-the-art' components, and thereby avoid the danger of confusing a system experiment with a device experiment.

3.1 Functional requirements

The choice of functions executed in the CAFS subsystem and the balance between those performed by hardware and those performed by software was of crucial importance if the resulting system were to have both adequate flexibility and attractive performance. The aim was to construct a filtering hierarchy in which the intrinsically high disc data transfer rate was handled by simple repetitive hardware, with progressively more complex operations being performed on successive abstracts of diminishing volume, culminating in procedures executed in the mainframe. If this were to be achieved it was necessary to identify functions which had a wide range of utility and independence of applications. Encouragement that this should be possible was obtained by noting the great success and widespread use of report generating packages such as ICL's FIND2. Many of these programs contained general purpose parametric routines capable of direct hardware implementation. Nevertheless, it did become clear that file structures would have to be kept reasonably simple if hardware complexity were to be contained. In practice this meant placing restrictions on the use of hierarchic records. However, this was not considered to be a serious deficiency, since, at the time, the project was being significantly influenced by the normalisation techniques of E.F.Codd in his proposed relational data base management system.[12]

The project commenced with an applications study from which it was concluded that the following functions were desirable in the disc store sub-system:

(*a*) Evaluation of record selection expressions that may involve nested boolean functions of many variables using the logical operators AND, OR, NOT.

(*b*) Evaluation of record selection expressions involving weighted threshold functions of many variables.

(*c*) Subsetting of selected records so as to return to the mainframe only those data items specified by the task generating process.

(*d*) Counting occurrences of records satisfying a selection expression.

(*e*) Using the relationships $\equiv$, $\not\equiv$, $>$, $<$, $\geqslant$, $\leqslant$, between specified key values and data item values as terms in a record selection expression.

(*f*) Masking of data item values to a resolution of at least a byte or character.

(*g*) Stem matching of individual terms.

(*h*) The use of maximum and minimum values of a data item value as a search term.

(*i*) Performing the summation of specified integer data item values in records satisfying a selection expression.

(*j*) The comparison of data item values of the same type to be available as a search term. This requirement was later augmented to include arithmetic operations involving addition and subtraction of data item values and literals.

(*k*) Evaluating selection expressions across virtual records formed by joining two or more physical files.

(*l*) Projection of a set of records satisfying a selection expression to remove redundancy with the option of counting the occurrences of each unique record.

The functions listed under (*a*) – (*g*) were implemented initially in a small experimental system which enabled a preliminary evaluation of the approach to be made. On the basis of the favourable results obtained a design was produced for a full scale disc controller incorporating special hardware for autonomous associative search.

3.2 Hardware

At the outset it was decided that the special features in the CAFS disc controller should be in addition to the standard direct access facilities of conventional equip-

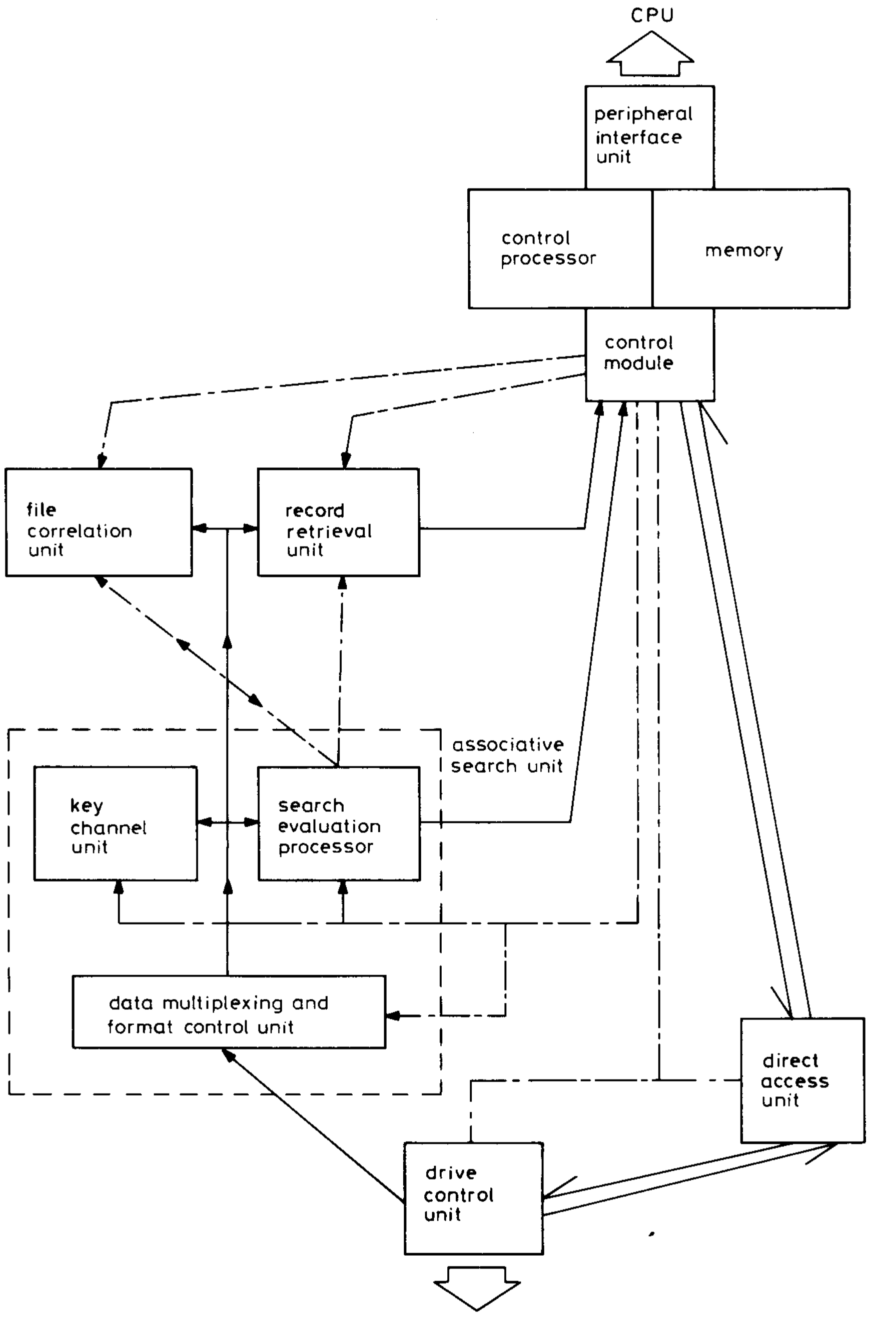


Fig. 1 CAFS controller functional subunits
main data paths
control paths

ment, i.e. block read and write. Within the controller there are six principal sub-units, namely:

(*a*) control processor
(*b*) direct access unit
(*c*) associative searching unit
(*d*) record retrieval unit
(*e*) file correlation unit
(*f*) drive control unit

The control processor is the 64 Kbyte machine taken from the ICL 7503 terminal controller. Its primary functions are task scheduling and resource management. The direct access unit, as its name implies, performs the standard functions of physical block reading and writing. The novel components in the system are the associative searching unit, the record retrieval unit and the file correlation unit. An outline block diagram is shown in Fig. 1.

The associative searching unit (ASU) is the heart of the system. Its function is to execute concurrent search tasks on a multiplexed data stream produced by the simultaneous reading of several disc channels. These channels may be allocated either separately or in groups to the disc drives. The drives themselves have additional read amplifiers so that several heads on any one may be read in parallel. Up to eight such multi-head read drives and six single head read drives may be connected to any one controller giving a total storage capacity of 840 Mbytes with EDS 60 drives. Within the ASU there are three principal subunits. These are the data multiplexing and format control unit, the key channel unit and the search evaluation processor. The first of these, as its name implies, takes the raw data from the multiple disc read channels and produces a single multiplexed output on a byte wide highway operating at 4 Mbytes/s. The present system will accept 12 individual disc channels of which up to ten may be allocated to a multi-head read drive for those tasks where intensive searching is required. The unit also issues format control information to the other units such as start of record, start of field, end of record etc.

The key channel unit permits up to sixteen key and mask registers together with corresponding comparators to be allocated to any task. Up to seven such tasks may run concurrently. These key matching channels, when loaded with appropriate key data and masks, operate simultaneously on the data stream, and for the record being scanned will register presence of key type, equivalence of key data, the inequalities 'less than', 'greater than', as well as all their logical inversions. These operations are performed 'on-the-fly' as it were; there is no block or track buffering. After each key channel has performed its specific comparison the result is stored and then subsequently used as an operand in a microprogram executed by the search evaluation processor when all key comparisons for that record have been made; i.e. when the hardware detects end of record. This processor is a small specially designed vector machine which is programmed specifically for each search task and may execute several search programs simultaneously. In order for the key channels to carry out their function the data stored on disc needs either to be in a fixed format or to be self-identifying. At the outset of the project a decision was made to adopt a format which permitted records to be of variable length and to contain multiple occurrences of variable length group fields. Each group field is preceded by an identifier code and a length and may consist of a set of fixed length data items followed by one variable length data item (Fig. 2). Any individual item, if required for key comparison, may then be isolated by means of a mask which is stored in the key channel together with the data. This facility, when applied to variable length items such as text words, enables stem matching to be easily implemented.

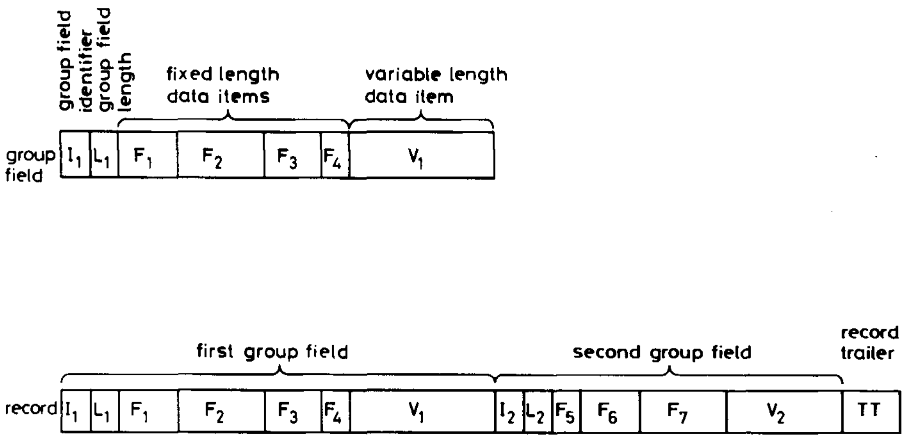


Fig. 2 CAFS record format

At the same time as the associative searching unit is carrying out its key comparisons the record retrieval unit is matching its identifier list against the data stream and collecting the contents of the designated fields in each record. If the record is subsequently declared to be a 'hit' these are returned; if not they are overwritten. This feature of hardware deblocking and editing of records is particularly valuable in interactive situations where high throughput is necessary. The data from the record retrieval unit is passed to the store of the control processor, where it may be further processed if required, before finally being returned to the user's work space in the mainframe. The further processing that could take place would, for example, be evaluating arithmetic selection expressions and summing integer field values.

The mechanism of selection is illustrated in Fig. 3. A search task, which may have come from a terminal enquiry or a batch program call and be of the form, 'GET NAME, PERSONNEL CODE FOR JOB = SALESMAN AND AGE <28 and BONUS> 750', is compiled by data management software held in the mainframe into a search and retrieve task specification. The latter, in the form of a list, contains key data, a microprogram for the search evaluation processor and a list of data items to be retrieved from selected records. This list is then passed by the operating system in the mainframe to the CAFS controller together with the physical addresses of the file areas to be searched. The size of these may vary from a single track to a whole disc cartridge depending on the extent of the file and the indexing strategy adopted. The control processor, having accepted the task, selects the appropriate disc drive and then transfers the task parameters to the relevant CAFS units.

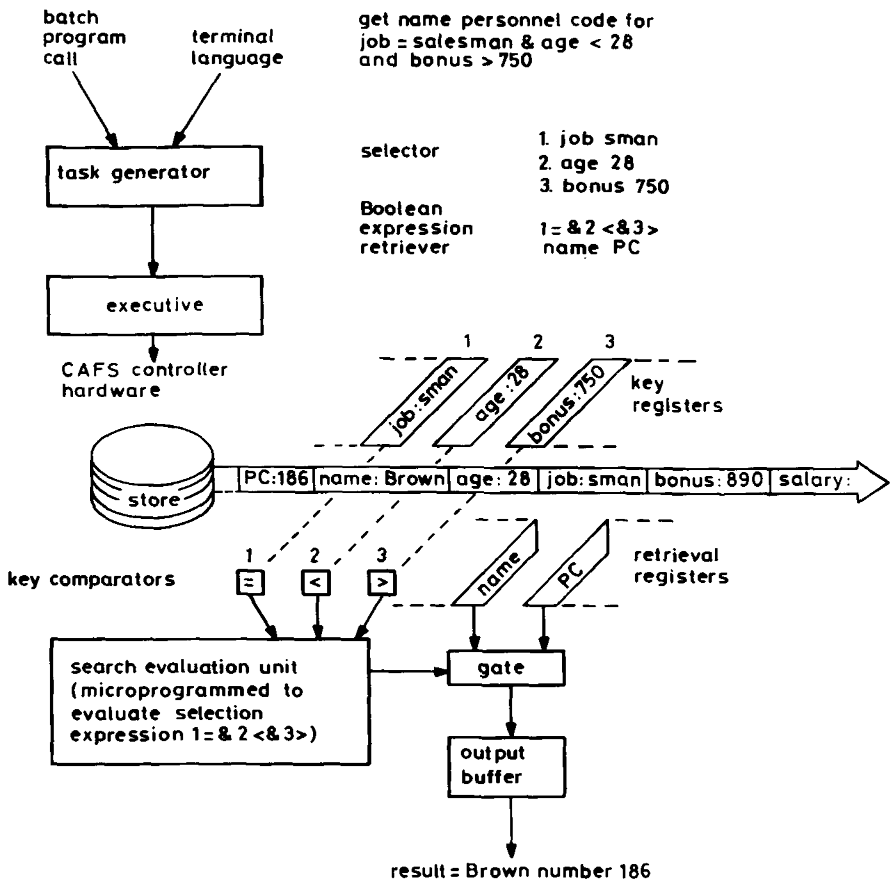


Fig. 3

A very valuable extension to the capability of the system is provided by the file correlation unit. This enables selection expressions to be evaluated on the joins of physical files without the overhead of sorting and merging intermediate results, or relying on precompiled pointer structures, provided that there exists a data item shared by records in the two files. The principal components of this device are a set of 256K 1-bit wide stores, any one of which may be addressed by the value part of a designated data item in the record being processed by the associative search unit. If the latter classifies a record as a 'hit' then the addressed bit can be set. At the end of a search task the store contains, in effect, the set of different data item values, coded as addresses, occurring in the designated field in all the records satisfying the search selection expression. On a subsequent search, on the same or a different file, the store containing these coded values may again be accessed during the processing of a record, but this time the state of the addressed bit may be treated as if it were a key channel comparator output by the microprogram performing the evaluation of the selection expression, thereby enabling a linked selection operation to be carried out across the two files. This is illustrated in Fig. 4. In this example, there are two files, a parts file and a supplier file, containing, respectively, part number, part description, supplier code and supplier name, supplier code, supplier address. Consider the enquiry, 'Find all suppliers of 1/4in Whitworth brass bolts in Birmingham'. To execute this, the parts file is first searched using 'part description ≡ 1/4in Whitworth brass bolts' as the selection expression. For all records found a bit in the map is set using the supplier code as an address. The supplier file is then searched using 'supplier address ≡ Birmingham and supplier code addressed bit set in map' as the selection expression. Supplier names are then retrieved from all records satisfying this expression.

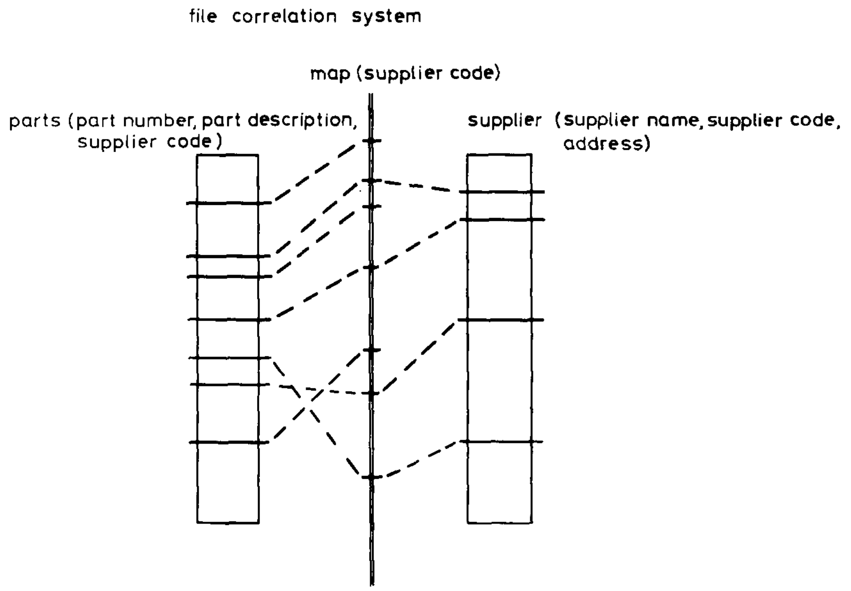

file correlation system
map (supplier code)
parts (part number, part description, supplier code)
supplier (supplier name, supplier code, address)

Fig. 4 'Find all suppliers of ¼ in Whitworth brass bolts in Birmingham'

For the record being processed the bit map store may be addressed directly by the value of the data item itself, by a numerical equivalent stored in the record or by a value obtained by hardware hashing. The latter, moreover, can be used on a virtual field assembled dynamically from more than one data item in the record. However, the use of hashing in this way can give rise to false 'hits' or 'ghosts', but the number of these can be reduced to almost negligible proportions by using several bit maps each of which is addressed via a different hashing function. Any residual ghosts are then removed by a backtracking operation.

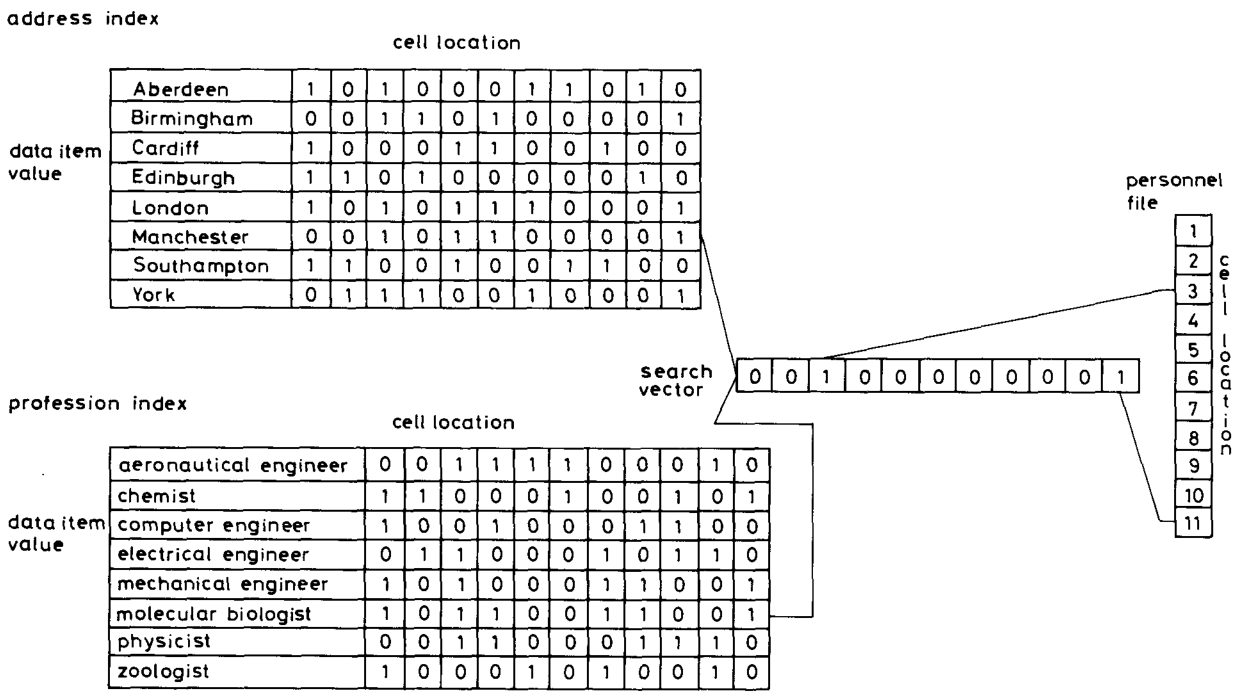
3.3 *File organisation and indexing*

The physical organisation of a CAFS file can be termed a cellular serial one. The file extent is divided into a series of storage cells whose size may vary from one disc track to a cylinder depending on the particular requirements of the application. Any search task is then directed to one or more cells which are exhaustively scanned. In many applications a half cylinder is used as a cell and then using 10-head read the whole cell may be searched in one revolution of the disc pack. Since access to a record within a cell is associative the physical location is irrelevant unless there is an applications requirement to maintain records in a given sequence. Although the CAFS hardware can provide very fast searching it is necessary to complement this by a low resolution indexing system in order to achieve the best performance. However, such indexes need only resolve to the level of storage cells. In many applications this means that each addressable unit, e.g. a half cylinder, may contain approximately 500-1000 records. There is then little difference between the time taken to select and retrieve one record from among a thousand others using CAFS than in making a random access to one using conventional methods. Indeed, in certain instances, the overall time for the latter could well be greater since additional disc accesses might well have to be made for index searching. Moreover, a CAFS file sequenced on a primary key would have only as many entries in the index as there were storage cells.

In addition, secondary or alternative key indexes can be readily compiled as an ordered list of values of a given data item in which each value has associated with it a binary vector having as many bits in it as there are storage cells in the file. Any particular bit in the vector is then set if there is at least one occurrence of this value in the corresponding storage cell. For a multi-factor selection expression, the low level system software can manipulate the index vectors of the various terms in the expression to produce a search vector. The bits that are set in this vector then correspond to the storage cells in the file in which it is worth making a search. This procedure is subtly different from conventional inverted indexing. Whereas the latter indicate where records *are*, the CAFS scheme indicates where they *are not*. This is illustrated in Fig. 5 for the simple enquiry on a personnel file. Assume that the file contains name, address, profession, and is ordered on name with secondary indexes for address and profession. Consider an enquiry such as, 'Find all molecular biologists in Manchester'. To execute this the secondary indexes would first be accessed and the vectors retrieved for 'address ≡ Manchester' and 'profession ≡ molecular biologist'. These vectors would then be intersected by the data management software in the mainframe to give a search vector containing bits set corresponding to the cells worth searching.

3.4 Creating and updating CAFS files

The physical mapping of a CAFS disc conforms to range standards, but, as already indicated, the format of records and their logical organisation within a file extent do not. Consequently, before a standard file can be read in CAFS search mode it has to be reloaded. Nevertheless, standard utilities can be used for disc initialisation, file allocation and file copying. Hard recording flaws are avoided by skipping the cylinder containing the flaw at the time of file allocation. Since the incidence of flaws on the 60 Mbyte cartridges used on the experimental system was extremely low, this simple minded approach to flaw management did not lead to a profligate waste of disc space.



address index

Aberdeen	1	0	1	0	0	0	1	1	0	1	0
Birmingham	0	0	1	1	0	1	0	0	0	0	1
Cardiff	1	0	0	0	1	1	0	0	1	0	0
Edinburgh	1	1	0	1	0	0	0	0	0	1	0
London	1	0	1	0	1	1	1	0	0	0	1
Manchester	0	0	1	0	1	1	0	0	0	0	1
Southampton	1	1	0	0	1	0	0	1	1	0	0
York	0	1	1	1	0	0	1	0	0	0	1

profession index

aeronautical engineer	0	0	1	1	1	1	0	0	0	1	0
chemist	1	1	0	0	0	1	0	0	1	0	1
computer engineer	1	0	0	1	0	0	0	1	1	0	0
electrical engineer	0	1	1	0	0	0	1	0	1	1	0
mechanical engineer	1	0	1	0	0	0	1	1	0	0	1
molecular biologist	1	0	1	1	0	0	1	1	0	0	1
physicist	0	0	1	1	0	0	0	1	1	1	0
zoologist	1	0	0	0	1	0	1	0	0	1	0

search vector

0	0	1	0	0	0	0	0	0	0	1

Fig. 5 CAFS secondary indexing

The updating of CAFS files can be performed purely conventionally using normal direct access methods or by using logical record insertion/deletion in CAFS mode. In the latter the mainframe software merely has to identify in which storage cell the insertion or deletion is to occur and then issue the appropriate commands. The CAFS controller will then perform the necessary block reading, repacking, writing back and check reading. In a transaction processing environment concurrent update control and jounalising for recovery remains entirely a mainframe data management function.

3.5 Performance

During the development of the CAFS system considerable emphasis was placed on pilot trials of applications already implemented conventionally. The results of these trials confirmed that substantial gains in performance were achievable over a wide variety of application types.

Table 1 lists some of these pilot implementations.

Table 1

Application	*Comments*
order book processing	mixed batch and TP
production control	mixed batch and TP
purchasing control	mixed batch and TP
keyword document retrieval	online enquiries and batch update
text retrieval	online enquiries and batch update
telephone directory enquiries	online enquiries and batch update
personal credit control	online enquiries and batch update
map cataloguing and retrieval	online enquiries and batch update
personnel record enquiries	online enquiries and batch update

To generalise the results obtained from these trials and attempt to quantify the overall gain is virtually impossible and would be of dubious value. In some situations, such as hashed random accessing of single records, there is hardly any difference at all, unless some advantage can be taken of concurrent reads on different disc drives. On the other hand, in batch mode multikey searching using programs such as FIND 2, reductions of 50-100 times in elapsed time and 10,000 times in mill usage are possible.

The latter is illustrated in Table 2 where the results are given for a serial search of a file containing all the commercial entries of the London telephone directory: a total of 250,000 records of some 23 million characters of loosely structured text. The search task was to match a single left-justified key stem of four characters against all words falling within the generic classification of name and business description and to provide a count of 'hit' records.

Table 2

Configuration	*Number of mainframe inst.*	*Elapsed time*	*Mill usage*
1903A + DA disc	148,000,000	16 min	~ 100%
1903A + CAFS	15,000	11 s	1%

Moreover, the CAFS hardware can execute a search using up to 16 key terms in a complex selection expression without there being any appreciable difference in the number of mainframe instructions and negligible difference in the elapsed time. If 16 keys had been used in the above example the number of mainframe instructions could well have been around 1000 million and the elapsed time would then be nearly 2 h.

Although the greatest performance gains can be achieved in the batch mode, it is with interactive enquiry systems that the processing power of CAFS can be most effectively used. The combination of high speed autonomous search and coarse indexing can enormously reduce the mainframe workload per terminal message-pair in many applications, and thereby enable a given mainframe to handle a substantially increased number of terminals with an acceptable response time. An illustration of this comes from the telephone directory enquiry application. An early analysis suggested that a software only system running on a 1904A might cope with one enquiry every two seconds, whereas a CAFS system on the same mainframe should manage eight enquiries per second. The recently completed Post Office trial established that this estimated performance for a CAFS system was indeed achievable. achievable.

4 Conclusions

The aim of the CAFS project was to demonstrate a new but essentially simple systems concept; namely, that by providing functionally specialised and dedicated processing units operating under the overall direction of mainframe software, but requiring only its minimal involvement, performance gains varying from one to two orders of magnitude could be obtained over a wide class of applications.

Such spectacular improvements could not have been achieved either by improving conventional software techniques or by increasing the raw power of the central processor. An appreciation of the limitations of von Neumann processing was required in order that a new balance be struck between hardware and software. Nevertheless these achievements would have been neither possible nor cost effective but for the dramatic developments in semiconductor technology over the last few years.

The potential processing power made available through the CAFS approach should have its major impact on the interactive environment, particularly in the field of information retrieval. Comprehensive language facilities may now be made available to a large number of users with response times such that a truly conversational and harmonious dialogue be established between man and machine.

Acknowledgments

The author is indebted to the Advanced Computer Techology Project of the Department of Industry which has supported a significant proportion of the work. He is also indebted to his colleagues at the Research and Advanced Development Centre, without whose industry and dedicated teamwork the project would not have been a success, and in particular to the late R.W.Mitchell who conceived the original idea.

References

1 BERRA, P.B.: 'Data Base Machines'. Proceedings of Infotech Conference, 'Database – The Next Five Years', December 1977.

2 SCARROTT, G.G.: 'Wind of Change', *ICL Tech. J.*, 1978, 1, pp.35-49

3 CODASYL DBTG 1971 Report. Conference on Data Systems Languages, ACM, New York.

4 COPELAND, G.P., LIPOVSKI, G.J. and SU, S.Y.W.: 'The architecture of CASSM: a cellular system for non-numeric processing'. Proceedings of the First Annual Symposium on Computer Architecture, December 1973.

5 OZKARAHAN, E.A., SCHUSTER, S.A. and SMITH, K.C.: 'RAP – associative processor for data base management', AFIPS Conference Proceedings, 1975, 44.

6 LIN, S.C., SMITH, D.C.P. and SMITH, J.M.: 'The design of a rotating associative memory for relational data base applications', *ACM Trans.* on Data Base Systems, March 1976, 1

7 LEILICH, H.O., KARLOWSKY, I., and ZEIDLER, H.C.: 'Content addressing in data bases by special peripheral hardware', Workshop on Computer Architecture, Erlangen, May 1975.

8 COULOURIS, G.F., EVANS, J.M. and MITCHELL, R.W.: 'Towards content addressing in data bases', *Comput. J.*, February 1972, 15.

9 MITCHELL, R.W.: 'Content addressable file store'. Proceedings of Online Conference on Database Technology, April 1976.

10 BABB, E.: 'Implementing a relational database by means of specialized hardware'. *ACM Trans.* on Database Systems 1979, 4, pp. 1-29

11 MALLER, V.A.J.: 'A content addressable file store'. IEEE Spring Computer Conference, San Francisco, 1979.

12 CODD, E.R.: 'A relational model of data for large shared data banks', *CACM*, June 1970, 13.

Computing in the humanities

Susan Hockey
Oxford University Computing Service

Abstract

This paper describes some of the ways in which computers are being used in language, literature and historical research throughout the world. Input and output are problems because of the wide variety of character sets that must be represented. In contrast the software procedures required to process most humanities applications are not difficult and a number of packages are now available.

1 Introduction

The use of computers in humanities research has been developing since the early 1960s. Although its growth has not been as rapid as that in the sciences and social sciences, it is now becoming an accepted procedure for researchers in such fields as languages, literature, history, archaeology and music to investigate whether a computer can assist the course of their research.

The major computer applications in the humanities are essentially very simple processes. All involve analysing very large amounts of data but do not pose many other computational problems, once the data is in computer readable form. The data may be a text for which the researcher wishes to compile a word index or alphabetical list of words, or it may be a collection of historical or archaeological material which the researcher wishes to interrogate or catalogue or from which he may perhaps wish to calculate some simple statistics.

2 Input and output

Getting the data into computer readable form is a major problem for any humanities application, not only because of its volume but because of the limited character set that exists on most computer systems. The original version of a text to be processed is more likely to be in the form of a papyrus, a medieval manuscript or even a clay tablet than a modern printed book. Many texts are not written in the Roman alphabet but, even if they are, languages such as French, German, Italian or Spanish contain a number of accents or diacritics which are not usually found on computer input devices. The normal practice is to use some of the mathematical symbols to represent accents so that for example the French word *Été* would be transcribed as *E*te**, assuming of course that the computer being used has upper and lower case letters.

A text which is not written in the Roman alphabet must first be transliterated into 'computer characters', so that for example in Greek, the letter alpha becomes *a*, gamma becomes *g* and not so obviously theta could be transliterated as *q*. In some languages such as Russian, it is more convenient to use more than one computer character to represent each letter. This presents no problems provided that the program which sorts the words into alphabetical order knows that its collating sequence may consist of such multiple characters. They are also necessary for modern languages such as Welsh and Spanish.

Arabic and Hebrew, though written from right to left in the original, are input in transliterated form from left to right. In neither language is it usual to write the vowel marks. Hebrew is easier in that it has only 22 letters, five of which have an alternative form when they appear at the end of a word. By contrast, each letter of the Arabic alphabet has four forms depending on whether it occurs in the initial, medial, final or independent position. Each of these forms can be transliterated into the same character, for it has proved possible to write a computer program to simulate the rules for writing Arabic, if the output should be desired in the original script.

There are of course a number of specialised input devices for non-standard scripts. The University of Oxford already has a visual display unit which displays Greek with a full set of diacritics as well as Roman characters. Other such terminals exist for Cyrillic, Hebrew and Arabic. Some, like the Oxford one, generate the characters purely by hardware, others use a microprocessor or even the mainframe computer to draw the character shapes. While this type gives a greater flexibility of character sets, it is slow to operate when its prime use is for the input of large amounts of data. Ideographic scripts such as Chinese, Japanese or even Egyptian hieroglyphs can be approached in a different manner. An input device is now in use for such scripts, which consists of a revolving drum covered by a large sheet containing all the characters in a 60 x 60 matrix. The drum is moved so that the required character is positioned under a pointer. The co-ordinates of that character on the sheet are then transmitted to a computer program which already knows which character is in that position.

Specialised devices are the only adequate solution to the problem of output. An upper- and lower-case line printer may be used at the proofreading stage but it does not have an adequate character set and the quality of its output is unsuitable for publication. A daisy-wheel printer may offer a wider character set and better designed characters but its printout resembles typescript rather than a printed book. There have been a number of experiments to use graphics devices to draw nonstandard characters. These have proved more successful than the devices mentioned above, particularly for Chinese and Japanese, but their main disadvantage is the large amount of filestore and processing time required to print even a fairly small text. It seems that the best solution to the output problem is a photocomposer which offers a wide range of fonts and point sizes and which can be driven by a magnetic tape generated on the mainframe that has processed the text. Such devices have been used with great success to publish a number of computer generated word indexes and bibliographies.

Besides simply retyping the text, there are other possible sources of acquiring a text in computer readable form. Once a text has been prepared for computer processing it may be copied for use by many scholars in different universities. There exist several archives or repositories of text in computer readable form which serve a dual purpose of distributing text elsewhere and acting as a home for material for which the researcher who originally used it has no further use. The LIBRI Archive at Dartmouth College, Hanover, New Hampshire, USA, has a large library of classical texts, mainly Latin. The Thesaurus Lingae Graecae at Irvine, California, USA, supplies Greek text for any author up to the fourth century AD. Oxford University Computing Service has an archive of English literature ranging from the entire Old English corpus up to the present day poets. The Oxford Archive will also undertake to maintain text in any other language if it is deposited there. The Literary and Linguistic Computing Centre at the University of Cambridge has a large collection of predominantly medieval text. All these archives maintain catalogues of their collections and will distribute text for a small charge. There has usually been no technical difficulty in transferring tapes from one machine to another, provided that variable-length records are avoided. Another source of text on tape is as a by-product of computer typesetting. A number of publishers have been able to make their typesetting tapes available for academic research after the printers have finished with them. These have been particularly valuable for modern material such as case and statute law as well as dictionaries and directories.

3 Applications

The most obvious text-analysis application is the production of word counts, word indexes and concordances. A word count is simply an alphabetical list of words in a text with each word accompanied by its frequency. In a word index each word is accompanied by a list of references such as chapter, page or line showing where that word occurs in the text. In a concordance each occurrence of each word is accompanied by some context to the left and to the right of it showing which words occur near it. The reference is usually also given. In many cases the context consists of a complete line of text, but it may be a whole sentence or a certain number of words to the right and to the left.

Sorting words into alphabetical order is not as simple as one might think. Most concordance programs, as they are usually called, recognise several different types of characters. Some may be used to make up words and can be called alphabetics. Some may be punctuation characters and are therefore always used to separate words. These two categories are fairly obvious, but consider the case where an editor has reconstituted a gap in an original manuscript and has inserted within brackets the letters which he considers are missing, for example the Latin word *fort[asse]*. In a concordance this word must appear under the heading *fortasse* therefore the brackets must be ignored when the word is alphabetised but retained so that the word appears in the context still in the form in which it appears in the text. Thus we have another category of characters, those which are ignored for sorting purposes but retained in the context. A fourth category is usually used for accents and diacritics and very often for apostrophe and hyphen in English. We can consider the simplest case, that of the French words *a*, part of avoir, and *à*, the preposition. The latter form must appear as a separate entry in the word list immediately after all the occurrences of *a*, but before any word beginning *aa-*. If the accent is treated as an alphabetic symbol it must be given a place in the alphabetic sequence of characters, but wherever it is placed it will cause some words to be listed in the wrong position. Therefore it is more usual and indeed necessary to treat accents as a secondary sort key which is to be used when words can no longer be distinguished by the primary key. The same treatment is necessary for an apostrophe and hyphen in English. There is no position for an apostrophe in an alphabetic sequence which will allow all the occurrences of *I'll* to appear immediately after all those of *ill* and all those of *can't* to come immediately after all those of *cant*.

The alphabetic sequence for sorting letters may vary from one language to another. Reviewers of concordances of Greek texts have been rightly scornful if the Greek words appear in the English alphabet order. A concordance program should be able to accept a user-supplied collating sequence of letters and operate using that. It should also be able to treat the double letters of Welsh or Spanish and the multiple characters of transliterated Russian as single units in the collating sequence. More importantly, it should have a facility to treat two or more characters as identical so that for example all the words beginning with upper case *A* do not come after all those beginning with lower case *a*, but are intermingled with them in their rightful alphabetic position. It follows then that the machine's internal collating sequence is not suitable for sorting text and it is regrettable that too many manufacturers rely on this collating sequence for their sorting software.

There are several different kinds of concordances and word lists. The most usual method is to list all the words in forward alphabetical order. They may also be given in frequency order beginning with either the most frequent or the least frequent. In this case when many words have the same frequency the words within that frequency are themselves listed in alphabetical order. Words may also be listed in alphabetical order of their ending, i.e. they are alphabetised working backwards from the ends of the words. This is particularly useful for a study of rhyme schemes or morphology (grammatical endings). In a concordance, all the occurrences of a word may appear either in the order in which they occur in the text or in alphabetical order of what comes to the right or left of the keyword, as it is called.

A vocabulary distribution of a text always gives a very similar sort of curve, ranging from a few very-high-frequency words down to a very long tail of many words occuring only one. The high-frequency words occupy a very large amount of room in a concordance and are therefore sometimes omitted. In the handmade concordances of the nineteenth century they were almost always omitted because they accounted for such a large proportion of the work. Words can be missed out either by supplying a list of such words or by a cutoff frequency above which words should not appear. A third option in a concordance is to give only the references and not the context for high-frequency words.

Foreign words, quotations and proper names can also cause problems in computer-genèrated concordances. If there are many of them and particularly if they are spelled the same as words in the main text, they can distort the vocabulary counts to such an extent that they become worthless. This problem is usually resolved by putting markers in front of these words as the text is input. For example, in a concordance of the Paston letters in preparation in Oxford, all the proper names are preceded by ↑ and all the Latin words by $. These markers can then be used to list all these words separately or if the editor wishes they can be ignored at the sorting stage so that the words appear in their normal alphabetic position.

Word counts and concordances are the basis of most computer-aided studies of text. Their uses range from authorship studies, grammatical analysis and phonetics to the preparation of language courses and study of spelling variations and printing. A few examples will suffice to show what can be done.

Vocabulary counts have been used in several ways for preparing language courses. The University of Nottingham devised a German course for chemists using vocabulary counts of a large amount of German technical literature as a basis for selecting which words and grammatical forms to teach. It was found, for example, that the first and second person forms of the verb very rarely appeared in the technical text, although they would normally be one of the first forms to be taught. A second approach in designing a language course is to use a computer to keep track of the introduction of new words and record how often they are repeated later on in the course.

A word list of a very large amount of text can give an overall profile of one particular language. Several volumes of a large word count of modern Swedish have been published, using newspapers as the source of material. Newspaper articles are an ideal base for such a vocabulary count as they are completely representative of the language of the present day. Another study of modern Turkish, again using newspapers, is aiming to investigate the frequency of loan words, mainly French and Arabic, in that language. By comparing two sets of vocabulary counts with a 5-year interval between the writing of the articles, it is possible to determine how much change has taken place over that period.

Some forms of grammatical analysis may also be undertaken using a concordance, particularly the use of particles and function words which are not so easy to notice when reading the text. In several cases the computer has found many more instances of a particle in a particular text than grammar books for that language acknowledge. It is also possible, albeit in rather a clumsy fashion, to study such features as the incidence of past participles in English. They may conveniently be defined as all words ending in *-ed*. This will of course produce a number of unwanted words such as *seed* and *bed* but they can easily be eliminated when the output is read. It is more important that no words should be missed and experiments have shown that the computer is much more accurate than the human at finding them.

Concordances can also be used to study the chronology of a particular author. It is certainly true that for a number of ancient authors, the order of his writings is not known. It may be that his style and particularly his vocabulary usage have changed over his lifetime. By compiling a concordance or word index of his works, or even by finding the occurrences of only a few items of vocabulary, it is possible to see from the references which words occur frequently in which texts. In this kind of study it is always the function words such as particles, prepositions and adverbs that hold the key, for they have no direct bearing on the subject matter of the work. These words have been the basis of a number of stylistic analyses of ancient Greek texts. In the early 1960s a Scottish clergyman named A.Q. Morton hit the headlines by claiming that the computer said that St. Paul only wrote four of his epistles. Morton has popularised the use of computers in authorship studies and at times tends to use too few criteria to establish what he is attempting to prove. His methods have, however, been taken up with some considerable success by a number of other scholars, notably Anthony Kenny, now Master of Balliol College, Oxford. Kenny used computer-generated vocabulary counts and concordances to study the Nicomachean and Eudemean Ethics of Aristotle. Three books appear in both sets of Ethics and these three books have been traditionally considered to be part of the Nicomachean Ethics and an intrusion into the Eudemean Ethics. By analysing the frequencies of a large number of function words, Kenny found that the vocabulary of the disputed books shows that they are more like the Eudemean than the Nicomachean Ethics.

Authorship studies are undoubtedly the popular image of text-analysis computing. There have been a number of relatively successful research projects in this area, but it is important to remember that the computer cannot prove anything; it can only supply facts on which deductions can be based. Mosteller and Wallace's analysis of the Federalist Papers,[1] a series of documents published in 1787-88 to persuade the citizens of New York to ratify the constitution, is a model study. There were three authors of these papers, Jay, Hamilton and Madison, and the authorship of 12 out of 88 was in dispute and that only between Hamilton and Madison, for it was known that they were not written by Jay. This was an ideal case for an authorship study for there were only two possible candidates and the papers whose authorship was known provided a lot of comparative material for testing purposes. Mosteller and Wallace concentrated on the use of synonyms and discovered that Hamilton always used *while* when Madison and the disputed papers preferred *whilst*. Other words such as *upon* and *enough*, for which there are again synonyms, emerged as markers of Hamilton. Working at roughly the same time as Mosteller and Wallace, a Swede called Ellegard applied similar techniques to a study of the Junius letters. Although he was not able to reach such a firm conclusion about their authorship, he and Mosteller and Wallace, to some extent together with Morton, paved the way for later authorship investigations.

There are other stylistic features besides vocabulary which can be investigated with the aid of a computer. In the late 19th century T.C. Mendenhall employed a counting machine operated by two ladies to record word-length distributions of Shakespeare and a number of other authors of his period. Although he found that Shakespeare's distribution peaked at 4-letter words, while the others peaked at three letters, he was not able to draw any satisfactory conclusion from this. It goes without saying that word length is the simplest to calculate but it is arguable whether it really provides any useful information. Sentence length has also been used as a criterion in stylistic investigation. This again has proved a variable measure, too often used because it is easy to calculate. Mosteller and Wallace first applied sentence-length techniques to Hamilton and Madison and found their mean sentence length was almost the same.

One feature which is more difficult to investigate with a computer is that of syntactic structure. Once one has gone further than merely identifying words which introduce, say, temporal clauses, it is not at all easy to categorise words into their parts of speech. Related to this question is the problem of lemmatisation, that is putting words under their dictionary headings. The computer cannot put *is, am, are, was, were* etc. under *be* unless it has been given prior instruction to do so. Most computer concordances do not lemmatise, but most of their editors would prefer it if they did. The most sensible way of dealing with the lemmatisation problem is to use what is known as a machine dictionary. This is in effect an enormous file containing one record for every word in the text or language. When a word is 'looked up' in the dictionary, its record will supply further information about that word such as its dictionary heading, its part of speech and also some indication if it is a homograph. The size of such a dictionary file can truly be enormous but it can be reduced for some languages if the flectional endings are first removed from the form before it is looked up. This is not so easy for English where there are so many irregular forms, but even removing the plural ending *-s* would reduce the number of words considerably. For a language like Latin, Greek or even German it is far more practical to remove endings before the word is looked up. For example in Latin *amabat* could always be found under a heading *ama*, which is not a true Latin stem, by removing the imperfect tense ending *-bat*.

A machine dictionary can therefore be used to supply the part of speech for a word. In the case of homographs such as *lead*, a noun, and *lead*, a verb, some further processing may be required on the sentence to decide which form is the correct one. Once the parts of speech have been found they can be stored in another file as a series of single letter codes and this file interrogated to discover, for example, how many sentences begin with an adverb, or what proportion of the author's vocabulary are nouns, verbs and adjectives.

Besides vocabulary and stylistic analysis, there are two more branches to text analysis where a computer can be used. We can consider first textual criticism, the study of variations in manuscripts. There exist several versions of many ancient and modern texts and these versions can all be slightly different from each other. With ancient texts, the manuscripts were copied by hand sometimes by scribes who could not read the material and inevitably errors occurred in the copying. When a new edition of such a text is being prepared for publication the editor must proceed through several stages before the final version of his text is complete. First he must compare all the versions of the texts to find where the differences occur. This technique is known as collation and the differences as variant readings which can be anything from one word to several lines. Once the variants have been found, the editor must then attempt to establish the relationship between the manuscripts, for it is likely that the oldest one is closest to what the author originally wrote. Traditionally manuscript relationships have been described in the form of trees where the oldest manuscript is at the top of the tree. More recently cluster-analysis techniques have been applied to group sets of manuscripts without any consideration of which is the oldest. The printed edition which the editor produces consists of the text which he has reconstructed from his collation and a series of footnotes called the apparatus criticus which consists of the important variants and the names of the manuscripts in which they occur.

The computer can be used in almost all these stages. It might seem that the collation of the manuscripts is the most obvious computer application, but this does have a number of disadvantages. The first problem is to put all the versions of the text into computer-readable form, a lengthy but surmountable task. More difficult problems arise in the comparison stage, particularly with prose text. In verse an interpolation or extra line must fit the metre. In prose it can be any length and most programs have been unable to realign the text after a substantial omission. If the manuscripts are collated by computer, the variants can be saved in another computer file for further processing. It is the second state of textual criticism, namely establishing the relationship between manuscripts, that a computer has been found more useful. Programs exist to generate tree structures from groups of variants, though most of them leave the scholar to decide which manuscripts should be at the top of the tree. Cluster-analysis techniques have also been used on manuscript variants to provide dendrograms or 3-dimensional drawings of groups of manuscripts. If one complete version of the text is in computer readable form it can be used to create the editor's own version and together with the apparatus criticus be prepared on the computer and phototypeset from it.

The other suitable text-analysis application is the study of metre and scansion in poetry, and to some extent also in prose. There are two distinct ways of expressing metre. Some languages like English use stress while others like Latin or Greek use the length of the syllable. In the latter case it is possible to write a program to perform the actual scansion; in the former it is more usual to obtain the scansions from a machine-readable dictionary. The rules of Latin hexameter verse are such that a computer program can be about 98% accurate in scanning the lines. The program operates by searching the line for all long syllables, that is those with a diphthong or where the vowel is followed by two consonants. The length of one or two other syllables, for example the first, can always be deduced by the position. It is usually then possible to fill in the quantities of the other syllables on the basis of the format of the hexameter line which always has six feet which can be either a spondee (two long syllables) or a dactyl (one long followed by two shorts). For languages like English, the scansion process may be a little more cumbersome but the net result is the same: a file of scansions which can then be interrogated to discover how many lines begin or end with particular scansion patterns.

Sound patterns of a different kind can be analysed by computer, particularly alliteration and assonance. Again the degree of success depends on the spelling rules of the language concerned. English is not at all phonetic in its spelling and simple programs to find two or more successive words which start with the same letter can produce some unexpected results. In other languages where the spelling is more phonetic this can be a fruitful avenue to explore. An analysis of the Homeric poems which consisted merely of letter counts revealed some interesting features. It was convenient to group the letters according to whether their sound was harsh or soft. When the lines that had high scores for harsh letters were investigated it was seen that they were about battle or death or even galloping hooves, whereas the soft sounding lines dealt with soothing matter like river or love. A comprehensive study of metre and alliteration was conducted by Dilligan and Bender[2] on the iambic poems of Hopkins. They used bit patterns to represent the various features for each line and applied Boolean operators to these bit patterns, for example to find the places where alliteration and stress coincided.

In contrast to raw text, historical and archaeological data are usually in a record and field structure. In fact many of the operations performed on this material are very similar to those used in commercial computing and many of the same packages are used. The files can be sorted into alphabetical order by one or more fields or they can be searched to find all the records that satisfy some criteria. If the material being sorted is textual, the same problems that occur for sorting raw text can also arise here. Another difference from commercial computing is the amount of information that is incomplete. It is quite common for a biographical record which can have several hundreds of fields to contain the information for only a very few. If the person lived several hundred years ago it is not likely that very much is known about him other than his name. Such is the case for the History of the University of Oxford [a Department of the University] who use the ICL FIND2 package for compiling indexes of students who were at the University in its early days. The computer file holds students up to 1560 and for each person gives such categories as their name, college, faculty, holy order, place of origin etc. The file is then used to examine the distribution of faculty by college, or origin by faculty etc. A related project of the History is an in depth study of Corpus Christi college in the seventeenth century. Here most of the information is missing for most of the people, but when the project is complete it is likely that some of the gaps can be filled in.

Historical court records have also been analysed by computer again to investigate the frequency of appearance of certain individuals and to see which crimes and punishments appear most frequently.

Archaeologists use similar techniques though their data more usually consists of potsherds, coins, vases or even temple walls. Another project in Oxford which now has about 50,000 records on the computer is a lexicon or dictionary of all the names that appear in ancient Greek, either in literary texts or inscriptions on coins or papyri. Each name has four fields which beside itself are the place where the person lived, the date when he lived and the reference indicating where his name was found. The occurrences of each name are listed in chronological order by place. Ancient dates are not at all simple. In this lexicon they span the period 1000 BC to 1000 AD approximately. Very few of the dates are precise. More often than not they consist of forms like 'possibly 3rd century' or 'in the reign of Nero' or 'Hellenistic'. When they are precise the ancient year runs from the middle of our year to the middle of the next year so that a date could be 151/150 BC. Sorting these dates into the correct chronological sequence is no easy task. It was solved by writing a program to generate a fifth field containing a code which could be used for sorting, but even after four years on the computer, more date forms are still being found and added to the program when required.

4 Software

Humanities users are reasonably well provided with packaged software. This is a distinct advantage in persuading them to use a computer because they frequently do not need to learn to program. The policy in universities is for the researchers to do their own computing but courses for both programming and using the operating system are normally provided and there is adequate documentation and advisory backup when it is needed. Most universities offer at least some of the humanities packages. For historical and archaeological work FIND2 has been used with considerable success. FAMULUS is a cataloguing and indexing package which is also used by historians and the Greek lexicon project mentioned above. It is really a suite of programs which create, maintain and sort files of textual information which are structured in some way. It is particularly suitable for bibliographic information where typical fields would be author, title, publisher, date, ISBN etc. It always assumes that the data are in character form so that the user must beware when attempting to sort numbers. In practice these are usually dates and can be dealt with in the same way as the lexicon dates. FAMULUS has a number of limitations, particularly that it only allows up to ten fields per record and that its collating sequence for sorting includes all the punctuation and mathematical characters and cannot be changed. It also sorts capital letters after small ones. There have inevitably been a number of attempts to rectify these deficiencies and versions have thus proliferated, but the basic function of the package remains the same. Its success is also due to the fact that it is very easy to use. Within the humanities it has been applied to buildings in 19th century Cairo, a catalogue of Yeats' letters, potsherds, court records as well as numerous bibliographies.

The advent of database-management systems such as IDMS could well bring some changes to humanities computing, but they are not easy to understand. The humanities user is almost always only interested in his results and it should be made as easy as possible for him to obtain these results. Work so far has shown that database users need a lot of assistance to start their computing but that there are considerable advantages in holding their data in that format. Experiments in Oxford with court records and with information about our own text archive have shown that the database avenue is worth exploring much further.

A number of packages exist for text-analysis applications, particularly concordances. At present the most widely used concordance package is COCOA which was developed at the SRC Atlas Computer Laboratory. COCOA has most of the facilities required to analyse text in many languages but its user commands are not at all easy to follow. In particular it has no inbuilt default system so that the user who wants a complete concordance of an English text must concoct several lines of apparent gibberish to get it. A new concordance package is now being developed in the Oxford laboratory which is intended to replace COCOA as the standard. It has a command language of simple English words with sensible defaults and incorporates all the facilities in COCOA as well as several others which have been requested by users. The package is being written in Fortran for reasons of compatibility. Fortran is the most widely used and widely known language in the academic world and it is hoped that a centre will be prepared to implement such a package even if it only has one or two potential users. To ensure machine independence, the package is being tested on ICL 2980, ICL 1906A, CDC 7600, IBM 370/168 and DEC 10 simultaneously. Tests will shortly be performed on Honeywell, Burroughs, Prime and GEC machines.

There are other concordance packages in the UK. CLOC, developed at the University of Birmingham, is written in Algol 68-R and has some useful features for studying collations or the co-occurrence of words. CAMTEXT at the University of Cambridge is written in BCPL and uses the IBM sort/merge and CONCORD written in IMP at Edinburgh is not unlike COCOA .

Other smaller text analysis packages are worth mentioning. EYEBALL was designed to perform syntactic analysis of English. It uses a small dictionary of about 400 common words in which it finds more than 50% of the words in a text. It then applies a number of parsing rules and creates a file containing codes indicating the parts of speech of each word. EYEBALL has a number of derivatives including HAWKEYE and OXEYE. All can give about 80% accurate parsing of literary English and higher for technical or spoken material. The only complete package for textual criticism is COLLATE, written in PL/1 at the University of Manitoba. This set of programs covers all the stages required for preparing a critical edition of a text, but it was written for a specific medieval Latin prose text and may not be so applicable to other material. Dearing's set of Cobol programs for textual criticism has proved difficult to implement on other machines. OCCULT is a Snobol program for collating prose text and VERA uses manuscript variants to generate a similarity matrix which can then be clustered by GENSTAT. One at least of the Latin scansion programs is written in Fortran and is in use in a number of universities.

Although Fortran is the most obvious choice of computer language for machine-independent software, it is not very suitable for text handling. Cobol is verbose and not widely used in the academic world and none of the Algol-like languages was designed specifically for handling text rather than numbers. Algol-68 has its advantages and supporters but it is not particularly easy for beginners to learn. The most convenient computer language for humanities research seems to be Snobol, particularly in the Spitbol implementation. It was designed specifically for text handling and with its patterns, tables and data structures it has all the tools most frequently needed by the text-analysis programmer. The macro Spitbol implementation written by Tony McCann at the University of Leeds and first available on 1900s has done much to enhance the popularity and availability of Snobol. It has also dispelled the belief, which arose from earlier interpretive implementations, that it is an inefficient language. But until there is a standard version of Snobol with procedures for tape and disc handling it is unlikely that it will ever have the use it deserves in text-analysis computing.

To sum up, then, there are ample facilities already available for the humanities researcher who wishes to use a computer. The software certainly exists to cover most applications. Even if the input and output devices leave something to be desired, it can be shown how much benefit can be derived from a computer-aided study.

Interest in this new method of research is now growing rapidly and it is to be hoped that soon all humanities researchers will take to computing as rapidly as their scientific counterparts.

REFERENCES

1 MOSTELLER, F., and WALLACE, D.L.: *Inference and disputed authorship* (Addison-Wesley, 1964)

2 DILLIGAN, R.J., and BENDER, T.K.: 'Lapses of time: a computer-assisted investigation of English prosody'. *in* AITKENT, A.J., BAILEY, R.W., and HAMILTON-SMITH, N. (Eds.): *The computer and literary studies* (Edinburgh University Press, 1973)

Erratum

ICL Tech.J., 1979, 1, p.180
The correct title of the paper by Prof. D.W.Barron is 'The next frontier: three essays on job control'

The data dictionary system in analysis and design

T J Bourne

ICL UK Division, Euston, London

Abstract

The Data Dictionary System is an information system for the data processing function. The paper introduces the subject with particular reference to ICL's data dictionary system (DDS). The need for a user oriented description of business information needs is discussed and an approach based on the use of the Data Dictionary System is outlined.

1 Introduction

The management and control of a data-processing department has never been easy. As well as the problems faced by the manager of any service department, the data-processing manager has to live with rapidly changing technology and an unusually high level of staff mobility. As a result, awareness has been growing in recent years of a number of common difficulties which adversely affect the efficiency and effectiveness of the data-processing activity. One approach which has been adopted by a number of users and software suppliers is to employ database techniques to provide an information system for the data-processing department, commonly known as the Data Dictionary System (DDS). This paper provides the background information on data dictionary systems in general and on ICL's DDS in particular, and describes an approach to systems design based on the use of a DDS.

The high cost of maintenance, typically amounting to well over half of the systems and programming budget, is a major problem for many organisations. The work commonly described as maintenance covers the correction of programming and design errors and the enhancement of an application system to provide additional facilities to its users; these latter range from minor operational conveniences to substantial changes to meet legal requirements, such as an increased number of VAT rates or a change in the method of calculating depreciation. The common factor in this work is that it is not a continuous job: the person doing it may have been on a different job yesterday and another tomorrow, and thus has to become familiar with the system before he can make a change to it. Good documentation is the key to this, and manually-maintained documentation is notoriously inadequate: it tends to be time consuming and expensive to keep up-to-date and difficult to use for *ad hoc* retrieval purposes.

A second area in which substantial problems arise is the co-ordination of distinct but related applications. It is not uncommon for what started as different applications to develop in such a way as to overlap, and this results in duplication of effort and inconsistent data. A large part of the problem here is that the process of systems design and analysis is carried out in many different ways, using many different types of documentation. Even where standards are laid down for this documentation it is very difficult to ensure that they are always followed in the same way.

These problems become even more apparent when an organisation adopts a policy of sharing a central database between a number of related applications. The requirement for a computer-based data-description library of some sort becomes so strong in this case that a number of users have made the implementation of their own data dictionary system their first application using a database management system.[1] The conclusion reached by users and suppliers alike is that the information needs of the data-processing department are at least as great as those of other departments such as production and accounting, and that these needs can best be met by a computer-based system. Such a system, usually known as a Data Dictionary System (DDS), would automate the storage and retrieval of information about applications and their implementation; the concept has significant benefits to offer to a data-processing department.

2 Background to ICL's DDS

The need for a data dictionary system was apparent from the inception of the 2900 series. Following a pilot study, work began on the present product in 1975 and it was made generally available early in 1977 after field trials from November 1976.

From the start there was active participation by prospective users in the specification of DDS and discussions continue with the user group which has now been formed. Another major influence on the development of the product has been the work of the Data Dictionary Systems Working Party (DDSWP) set up by the British Computer Society.[2] This group, composed of a representative mixture of users, research workers, consultants and suppliers, has done much valuable pioneering work in identifying potential areas of application of data dictionary systems and in outlining the facilities which such systems can be expected to provide. In particular, the working party emphasises that a complete system must go much further than simply documenting the data as it is held in the computer; this point will be developed further in the following Sections.

3 Features of ICL's DDS

The DDS meets the needs outlined previously by providing a single integrated database for the data-processing function. This covers not only computer data, such as records, files and IDMS databases but also the business view of data, describing things such as orders and deliveries. Furthermore, it includes descriptions both of computer processes, such as programs and modules, and of the corresponding business processes such as order processing, paying staff and so on. This dictionary database is shown diagrammatically in Fig. 1.

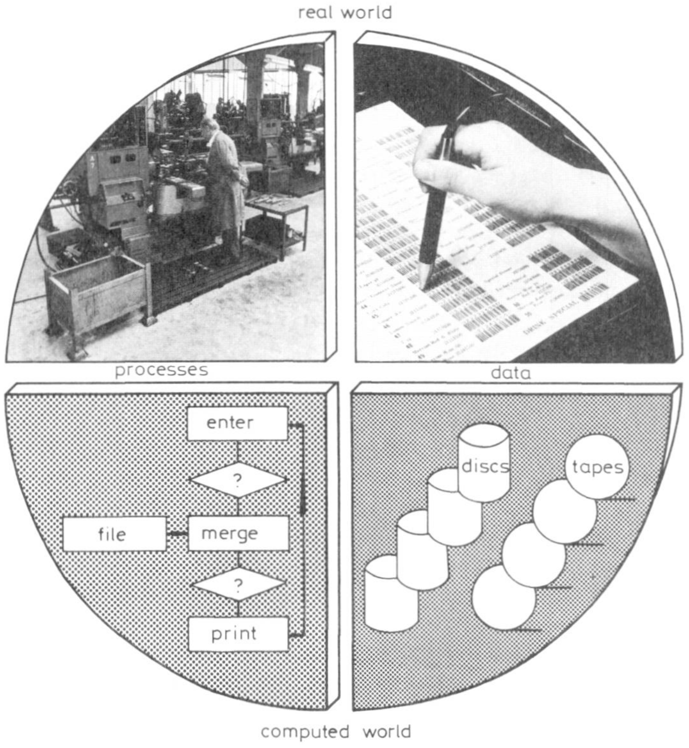


Fig. 1 Dictionary database

It is not enough, however, to provide a comprehensive database. The system also provides flexible facilities for input and output and a very wide range of reports, including the sort of cross-reference reports which could never be provided by manual documentation. Where data descriptions are required by other products, e.g. IDMS (Integrated Database Management System) or Cobol, they can be conveniently taken from the dictionary. The use of the database software IDMS by DDS means that any item of information about data or their use may be accessed in different ways for different purposes. This means that duplication can be greatly reduced or even eliminated. For example, a data item such as 'PART NUMBER' can be described just once, with its properties such as text description, permissible values and so on, and then referred to in many different record descriptions. An inquiry can then produce a list of all the records or files in which the item appears.

To facilitate its use by development teams working separately but on related projects, the system allows users or groups of users to be defined. Each such user has full control over any information which he entrusts to the system but can permit others to have access to it where necessary. Even where a dictionary is used by only a single project it may be necessary to record the changing state of a system during a phased development by holding several versions of data and program descriptions; DDS allows this also. To permit additional classification of the information held in the dictionary according to the needs of a particular organisation, each element (record, program, data item etc.) may be given a number of classification keys; these keys may then be used to qualify subsequent enquiries, thus restricting the amount of information retrieved.

4. The statement of business needs

The documentation of a computer-based information system has, in the past, tended to consist mainly of a description of the computer programs and computer-held data, together with a statement of the inputs to the system (forms etc.) and the outputs from it (reports, display screen formats etc.). Although other documents of various sorts have no doubt been used in the system design and analysis process, they have seldom been considered a part of the ultimate system documentation. A number of attempts have been made in recent years to improve the position, either by providing formal means for describing the original business situation or by separating the functional design from the details of a particular computer implementation.[2,3,4] Such developments can yield a number of benefits: in particular, it should be possible to define the business requirement in terms which can be fully understood by the user who has no data-processing background. The ICL approach is described in some detail below; first it may be helpful to outline some of the more significant influences on this aspect of DDS.

In their Interim Report[3] the ANSI/X3/SPARC DBMS Study Group put forward the case for a 'Conceptual schema'. This was to be a definition of the content and logical structure of a database biased neither to the needs of particular applications nor to the constraints of a particular implementation. This idea has been discussed extensively and in considerable detail by the ISO Working Group on DBMS (ISO TC97/SC5/WG3), but no detailed proposal has yet emerged and there is little prospect of standards in this area being defined for some years to come.

From its inception the aforementioned BCS Data Dictionary Systems Working Party saw an immediate need for facilities 'to record and analyse requirements independently of how they are going to be met'. It developed the notion of a 'conceptual view which describes the nature of the enterprise and its data in terms which are quite independent of any data-processing implications'. This conceptual view would constitute a model of the enterprise, describing the things of interest to it, the functions it can perform and the events which influence its behaviour. The facilities provided in ICL's DDS for documenting the 'real world' (Fig. 1) have evolved as a result of ICL's participation in the DDSWP and provide an interesting example of the potential value of such a group.

5 An approach to systems analysis

For the purposes of this paper, systems analysis can be defined as the process of investigation, study and documentation which results in a model of an enterprise and its data which are sufficient for some stated objective. This objective may be the design and implementation of one or more computer-based applications or simply to gain a better understanding of the enterprise and how it works. In the context of the Data Dictionary System, then, the output of a systems analysis exercise will be a model expressed in terms of the 'real world' half of the DDS, in sufficient detail for the stated objective.

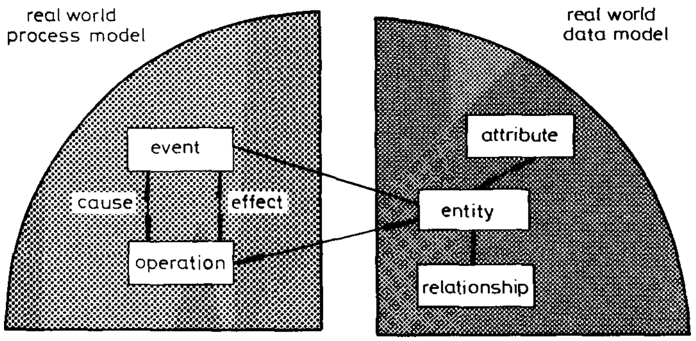


Fig. 2 Five concepts to describe an enterprise and its associated activities, interactions and information

The terms used by the DDS to describe the real world may be new to some but are those which are becoming widely accepted. Experience has shown that the following five concepts are sufficient to describe an enterprise, the activities which must go on within it, the way it interacts with the world around it and the information with which it is concerned (Fig. 2).

Entity	An entity is any thing, person, place, event or concept which can be identified as being of interest to the enterprise and about which information may arise or be required, e.g. employees, parts, sales areas and orders may all be entities
Attribute	An attribute is any property of an entity which is of interest, e.g. the name of an employee, the weight of a part or the value of an order
Relationship	A relationship is a connection between two entities, such as the fact that a particular employee works in a particular department. It is generally possible to represent such a relationship in terms of two entities having a common attribute: e.g. one may treat the department in which an employee works as an attribute of the employee

	and thus represent the relationship implicitly. However, in practice it has been found helpful to represent relationships explicitly and DDS therefore allows either approach
Operation	An operation is an activity which changes the state of the enterprise, by changing values of attributes, creating new entities and so on, or which transfers information between the enterprise and the outside world. Examples of operations are the allocation of stock to meet an order, the processing of a stated increase in staff salaries or the production of the annual company report. It is often convenient to define an operation initially at a fairly high level and subsequently break it down into suboperations as the analysis proceeds. The DDS allows operations to be broken down in this way to as many levels as necessary
Event	An event is something which happens, within the enterprise or in the outside world, which triggers one or more operations and may be the result of the completion of an operation, e.g. the arrival of an order is an event, triggering validation of the order, allocation of stock and so on.

How are these concepts to be used in building a model of an enterprise and its data? There are of course many possible approaches; one which is recommended by ICL is described in Reference 5. What follows is an illustration of the key points, intended particularly to demonstrate how the concepts are used in practice.

Consider the publication of an issue of this (or any) journal and in particular the selection of the papers it is to contain. In what types of entity are we interested? Some obvious ones are: issues of the journal, papers, authors, referees. In each of these cases each individual entity (a paper, an author) is easy to identify, by date, title or name, so we know what are the entities in which we are interested. The question is: how does each enter our field of interest? What is the operation which brings the entity into being? It is often only by examining this question that the real meaning of the entity can be made clear. For example, consider a paper. It becomes of interest when the editor becomes aware of its possible existence, not, as we might have supposed at first thought, when it is written. In fact there are probably two possible operations which can create the entity 'paper' in the model: 'commission a paper' and 'consider offer of a paper'. Having established some of the main entities and operations in this way, we can now look for the events which trigger these operations and for the information we need about each entity, in terms of attributes and relationships. For example, for an author we should need name, address and telephone number, possibly something on his professional position or status and whether or not we had previously considered any papers from him. The first four would be attributes and the last would be represented by relationships with papers about which we already had information.

The DDS is designed for interactive use, and it is quite permissible to enter partial definitions to be completed later, or to include reference to definitions which are yet to be entered. The following example shows how some of the information discussed above could be entered:

```
INSERT
ENTITY PAPER
*ATTRIBUTES TITLE, LENGTH, AUTHOR-NAME
*IDENTITY TITLE
ENTITY AUTHOR
*ATTRIBUTES AUTHOR-NAME, ADDRESS, PHONE
*IDENTITY AUTHOR-NAME
OPERATION CONSIDER-OFFER
*ENTITIES PAPER, AUTHOR
ATTRIBUTE LENGTH
* UNITS WORDS
* NOTE NORMALLY ABOUT 12 WORDS PER LINE
```

The commands for retrieval are equally simple, e.g. to find all the entities with the attribute AUTHOR-NAME, the DDS user would enter

```
FOR ATTRIBUTE AUTHOR-NAME
ENQUIRY ALL ENTITY
```

This brief and simple example should indicate the value of the technique in arriving at a clear understanding of an area of interest. The ability to use the DDS to document such work, and to obtain answers quickly to questions which would otherwise have involved much searching through files of forms, can transform the job of the systems analyst. It has the further great benefit of making his work readily accessible to other analysts, as well as to the users for whom he is working.

6 Summary

In view of the present combination of rising staff costs and falling hardware prices, any data-processing manager must surely be interested in any approach which can use assistance from the computer to increase the productivity of his staff. The Data Dictionary System offers this assistance, not only in development and maintenance of computer-based applications, but also in the earlier stages of systems analysis and design. As is common when the database approach is adopted, the existence of this database in the data processing department can be expected to open the way to other related applications using the same data.

References

1 GRADWELL, D.J.L.: 'Why data dictionaries?, *Database Journal,* 6, p.2

2 BRITISH COMPUTER SOCIETY: Report of the Data Dictionary Systems Working Party, March 1977

3 ANSI: Interim Report of the Study Group on Data Base Management Systems, ANSI/X3/SPARC, 1975

4 TEICHROW, D. and WINTERS, E.: 'Recent developments in system analysis and design', *Atlanta Economic Review*, 1976, Nov-Dec

5 ELLIS, H.C.: 'Analysing business information needs,' Conference Proceedings on Data Analysis for Information Systems Design BCS, 1978, June

Notes for authors

1 Content

The *ICL Technical Journal* publishes papers of a high technical standard intended for those with a keen interest in and a good working knowledge of computers and computing, but who nevertheless may not be informed on the aspect covered by a given paper.

The content will have some relevance to ICL's business and will be aimed at the technical community and ICL's users and customers. It follows that to be acceptable, papers on more specialised aspects of designs or applications must include some suitable introductory material or references.

The Journal will usually not reprint papers already published, though this does not necessarily exclude papers presented at conferences. It is not necessary for the material to be completely new or original (but see 10, 12 and 13 below). Papers will not reveal matter related to unannounced ICL Products.

2 Authors

Anyone may submit a paper whether employed by ICL or not. The Editor will judge papers on their merits irrespective of origin.

3 Length

Full papers may be of up to 10 000 words, but shorter papers are likely to be more readily accepted. Letters to the Editor and reviews may also be published.

4 Typescript

Papers submitted should be typed in double spacing on one side of A4 paper with full left-hand margin. Mathematical expressions are best written in by hand. Care should be taken to form Greek letters or other unusual symbols clearly. Equations referred to in the text should be numbered. Detailed mathematical treatments should be placed in an Appendix, the results being referred to in the text.

At least two copies should be submitted, both carrying the author's name, title and date of submission.

5 Diagrams and tables

Line diagrams supplied will if necessary be redrawn before publication. Be especially careful to label both axes of any graphs, and mark off the axes with values of the variables where relevant.

All diagrams should be numbered and supplied with a caption. The captions should be typed on a separate sheet forming part of the manuscript. Since diagrams may have to be separated from their manuscript every diagram should have its number, author's name and brief title on the back.

All diagrams and Tables should be referred to in and explained by the text. Tables as well as diagrams should be numbered and appear in the typed MS at the approximate place, at which they are intended to be printed. Captions for Tables are optional. Be careful to ensure the headings of all columns in Tables are clearly labelled and that the units are quoted explicitly in all cases.

6 Abstract

All papers should have an abstract of not more than 200 words. This ought to be suitable for the various abstracting journals to use without alterations.

7 Submission

Before submission authors are strongly urged to have their MSS proof read carefully by a colleague, to detect minor errors or omissions; experience shows that these can be very hard for an author to detect. Two copies of the MS should be sent to the Editor.

8 Referees

The Editor may refer papers to independent referees for comment. If the referee recommends revisions to the draft, the author will be called upon to make those revisions. Minor editorial corrections, e.g. to conform to a house style of spelling or notation, will be made by the Editor. Referees are anonymous.

9 Proofs

Authors will receive printed proofs for correction before publication date.

10 References

Prior work on the subject of any paper should be acknowledged, quoting selected early references. It is an author's reponsibility to ensure references are quoted; it will be unusual for a paper to be complete without any references at all.

11 Style

Papers are often seen written in poor or obscure English. The following guidelines may be of help in avoiding the commoner difficulties.

- Be brief.
- Short sentences are better than long ones but on the other hand do not write telegrams.
- Avoid nested relative clauses; preferably start new sentences.
- Define the meaning of ordinary words used in special senses. Define acronyms or sets of initials by quoting the full meaning the first time the initials are mentioned.
- Include a glossary of terms if necessary.
- Avoid words in brackets as much as possible.
- Avoid the frequent use of the type of construction known as a 'buzzword'. This often takes the form of a noun followed by a present or past participle followed by another noun e.g. 'system controlling parameters'.
- Take care in using the word 'it' that the reader will easily understand what 'it' refers to. An unambiguous rule, that cannot always be applied, is that 'it' refers to the nearest preceding noun in the singular.
- Several 'its' in one sentence each used in a different sense can cause considerable confusion. Similar remarks apply to 'this', 'that' and other prepositions.

12 Copyright

Copyright in papers published by the ICL Technical Journal rests with ICL unless specifically agreed otherwise before publication. Publications may be reproduced with permission and with due acknowledgement.

13 Acknowledgements

It is customary to acknowledge the help or advice of others at the end of papers when this is appropriate. If the work described is not that of the author alone it will usually be appropriate to mention this also.