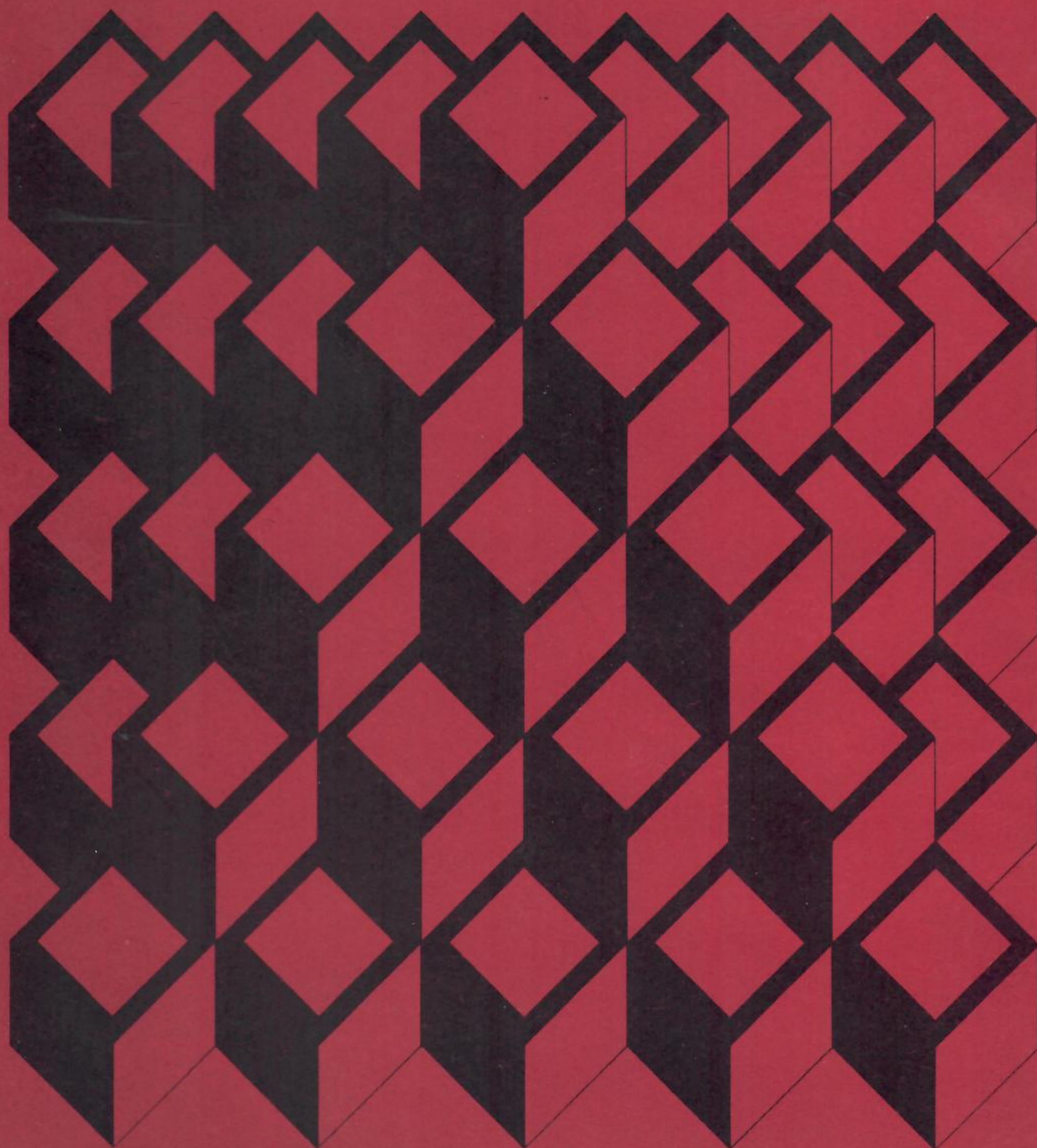
ICL

Introduction to GEORGE 3

1900 Series

COPY No. 001 OF
LIBRARY ACCESSION No. 710573



ICL

Introduction to GEORGE 3

1900 Series

The policy of International Computers Limited is one of continuous development and improvement of its products and services, and the right is therefore reserved to alter the information contained in this document without notice. ICL makes every endeavour to ensure the accuracy of the contents of this document but does not accept liability for any error or omission. Any equipment or software performance figures and times stated herein are those which ICL expects to be achieved in normal circumstances. Whereever practicable, ICL is willing to verify upon request the accuracy of any specific matter contained in this document.

Technical Publication 4265

© International Computers Limited 1971

Second Edition April 1971

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

Preface

GEORGE 3 is the full ICL operating system for all unpaged central processors in the 1900 Series from the 1903A upwards. References to "the current mark" are to Mark 6 of GEORGE 3. The manual will be brought up to date as further marks of the system are issued.

The minimum configuration for GEORGE is:

- a 1903A central processor with the following features:
  - extended mode
  - a real time clock
  - a program timer
  - the commercial computing feature
- 48K words of core store if programs are small; otherwise 64K is a reasonable minimum
- 512K words of direct access backing store (storage units and/or drum)
  - four magnetic tape decks
  - a line printer
  - a paper tape reader or card reader

The extended mode feature is required even with a core store of 32K words.

If MOP is to be used then 7071 teletypewriter consoles are required. These can be connected via:

- a 7007/2 multiplexer and 7008 data terminals
- a 7920 or 7930 scanner
- a 7900 communications processor and 7920 or 7930 scanner

Alternatively, if only a few consoles are being used, then each console can be connected via a 7070 data terminal.

If remote batch processing is to be used then 7020 terminals on telephone lines are required. These can be connected via:

- 7010/3 or 7010/5 or 7010/7 uniplexers
- a 7007/2 multiplexer and 7010/1 or 7010/4 or 7010/6 data terminals
- a 7920 or 7930 scanner
- a 7900 communications processor and 7920 or 7930 scanner

Each 7020 terminal must have a 7023 teletypewriter connected to it.

A reasonable configuration for MOP would include 96K words of core store, a drum, a storage unit, and ten to twenty consoles.

This manual is directed at users who have no experience of GEORGE 3 but posses a basic knowledge of 1900 Series programming. It is essential that users read *Introduction to GEORGE 3* before tackling the main system manual, *Operating Systems GEORGE 3 and 4* (4th Edition, TP 4267).

Introduction to GEORGE 3 contains seven chapters, of which the first is a general introduction to operating systems. The second chapter covers the GEORGE command language and the function of the job description in a GEORGE environment. Chapter 3 deals with backing store and includes a section of the use of the editing facility. Chapter 4 introduces methods of handling input and output to user programs. Chapters 5, 6 and 7 describe MOP (Multiple On-line Programming), scheduling, and budgeting facilities respectively. Appendix 1 describes the organization of a job description.

In addition to this introductory manual and the system manual, *GEORGE 3 Operating* (Edition 2, TP 4223) and *GEORGE 3 and 4 Operation Management* (Edition 3, TP 4199) describe the functions of the operator and the installation manager of an installation running under GEORGE 3. More detailed information on the use of MOP is available in the manual, *Introduction to MOP* (Edition 1, TP 4194).

This edition of the *Introduction to GEORGE 3* supersedes all previous editions.

Contents

Preface iii

Chapter 1 Introduction to operating systems 1
OPERATING SYSTEMS 1
GEORGE 3 1
Job descriptions 2
Internal management of central processor and core store 2
Remote batch processing 3
Backing store and off-line facilities 3
MOP (Multiple On-line Programming) 4
Work scheduling 4
Logging and accounting 5
GEORGE 4 5
OTHER ICL OPERATING SYSTEMS 5

Chapter 2 Jobs 7
COMMAND LANGUAGE 7
Format of commands 7
Built-in commands 7
Macro commands 7
System macros 8
User macros 8
Command processor levels 8
Parameters in user-defined macros 9
Conditional commands 9
Program event messages 10
Types of job description 10
Monitoring files 10
INPUT/OUTPUT FOR OBJECT PROGRAMS 11
Input and output using off-lining facilities 11
Input and output using on-line peripherals 11
TYPES OF JOBS 11
On-line jobs 11
Background jobs 12

Chapter 3 Backing Store 13
THE ENTRANT CONCEPT 13
The contents of an entrant 13
THE FILESTORE 13
Device independence 13
Types of file 13
Modes of access 14

The structure of the filestore 15
Filenames 17
Reasons for the structure of the filestore 18
THE BACK-UP SYSTEM 19
THE EDITOR 21
Calling in the editor 21
Editing language 21
Transcribing: T 23
Positioning the pointer: P 24
Inserting: I 25
Replacing data: R 26
The Forget (F) instruction 26
Ending the edit: The E and Q instructions 26
ENTRANTS OUTSIDE THE FILESTORE 27
Secure entrants 27
Insecure entrants 27
Conversion of entrant categories 29

Chapter 4 Input/output facilities 31
INPUT TO THE FILESTORE 31
The INPUT command 31
Embedded INPUT commands 31
Embedded data 32
INPUT AND OUTPUT FOR OBJECT PROGRAMS 32
Off-line peripherals: ASSIGN and LISTFILE 32
On-line peripherals 34
THE PROPERTY SYSTEM 34
The PROPERTY and ATTRIBUTE commands 35
Some uses of the property system 35
MULTIPLEXERS 35
7020 REMOTE TERMINALS 35
Facilities provided on 7020 terminals 35
PERIPHERAL CLUSTERS 36
THE OPERATOR'S FUNCTION 36
MOP TERMINALS AS INPUT/OUTPUT DEVICES 36

Chapter 5 MOP 37
INTRODUCTION 37
ENVIRONMENT 37
SYSTEM CONTROL OF ON-LINE USE 37
THE BREAK-IN FACILITY 38
Break-in levels and command processor levels 38
MONITORING WITH MOP 38
TYPICAL MOP OPERATIONS 38
FURTHER MOP FACILITIES 39
Background jobs 39
Conversing with an object program: The ONLINE command 40
Subsystems under MOP 40
EXAMPLE OF A MOP JOB 40

4265(4.71)

Chapter 6 Scheduling 43
SCHEDULING 43
THE HIGH LEVEL SCHEDULER 43
Factors relevant to high level scheduling 43
What the high level scheduler does 44
THE LOW LEVEL SCHEDULER 44
EXECUTIVE SCHEDULING 45

Chapter 7 Budgeting and accounting 47
BUDGETARY CONTROL 47
CLASSIFICATION OF BUDGETS 47
Transient budgets 47
Stable budgets 48
ALLOCATION OF BUDGETS 48
REMOVAL OF BUDGETS 48
PRIVILEGES 48

Appendix 1 Writing a job description 49
THE COMPLETE JOB DESCRIPTION 49
INTRODUCING THE JOB DESCRIPTION TO GEORGE 50
THE WHENEVER COMMAND 50
COMPILING THE SOURCE PROGRAM 50
CONNECTING PERIPHERALS TO THE PROGRAM 50
ENTERING THE PROGRAM 50
PROGRAM EVENTS 51
THE PRINT COMMAND 51
THE RESUME COMMAND 51
THE RUNJOB COMMAND 51

Index 53

Illustrations

Figure 1	Filestore structure	16
Figure 2	Company Structure 1	18
Figure 3	Company Structure 2	18
Figure 4	Entrant category conversion	28

Chapter 1 Introduction to operating systems

Computers consist of *hardware*, the electronic machinery by which data processing is carried out, and *software* which activates and directs the hardware. Software is the collective term for programs, the sequences of instructions which, translated into a computer readable form, actually control the computer's activities. Programs fall into three categories:

1 Those designed by user programmers, or possibly supplied by the manufacturer, to solve the particular problems presented in the applications for which the computer was acquired.

2 The *compilers*, programs which translate user programs written in programming languages into machine code.

3 The *control programs* and *operating systems* which control the running of other programs.

Compilers, control programs and operating systems are always supplied by the manufacturer, and are as indispensable as the hardware in making the computer work.

An example of a control program is Executive, permanently located in the core store of the ICL 1900 Series computers. Executive regulates the flow of work through the computer in conjunction with the human operator. It handles the transfer of data between peripheral (input and output) devices and the core store.

Executive checks that the peripherals are functioning correctly and informs the operator when they require attention. It translates and implements commands input by the operator via the console typewriter. Executive, except for small configurations, provides multiprogramming facilities; where these are available, Executive allocates peripherals as required, and selects programs to be run in accordance with the priorities specified by the programmer. Executive also handles *extracode* facilities. (Extracodes are routines that supplement the standard hardware order code. In some small central processors, for example, multiplication and division must be carried out by extracodes.)

Despite Executive's internal organizing powers, human intervention is still necessary to run a computer. The operator is responsible for servicing the basic peripherals; supplying the line printer with paper and the card punch with cards, for example. It is the operator who initiates a program run, and who is largely responsible for the scheduling when multiprogramming.

Executive, the hardware and the operator together form the basic *operational environment* of a 1900 Series installation, with the operator as the point of contact between the operational environment and the user.

OPERATING SYSTEMS

In a small installation, the combination of the operator and Executive is usually sufficient to keep the work flow up to the required level. However, in a large installation where many programs are being run, the operator's activities become time-consuming to the point where it is impossible to obtain maximum efficiency. An *operating system* is designed to help the user with a large installation make the most of his machine's potential. This amounts to cutting down as far as possible on the need for the operator to intervene in the running of programs.

The objectives of an operating system are:

1 *As an aid to management*, to increase the overall efficiency of an installation by increasing the throughput rate and by providing accurate budgeting and accounting facilities.

2 *As an aid to programmers* in the development of programs, to decrease the turnround time of a program, eliminate operator error and improve program testing and postmortem facilities.

3 To provide *extra facilities* not available in the basic operational environment, for example MOP (Multiple On-line Programming, see below, and Chapter 5) and remote batch processing (see below).

GEORGE 3

The 1900 Series Executive could be considered as a small operating system.

However, the term 'operating system' usually refers to a set of software routines exercising much more extensive control over the computer's activities than Executive does. Operating systems are designed to provide extra facilities which help to achieve the objectives already defined. The GEORGE 3 operating system achieves these objectives by providing the following facilities:

1 Automatic handling of operating instructions which would normally have to be carried out by the operator in the course of the run.

2 Efficient internal management in the use of core store and the central processor.

3 Giving programmers direct (on-line) access to the computer, bypassing the operator.

4 Batch processing using peripherals remote from the machine room.

5 Software control of files and a security system for controlling access to them.

6 Minimizing reliance on basic peripherals by off-lining to magnetic media.

7 Automatic work scheduling.

8 Accurate and flexible budgeting and accounting facilities.

The next sections of this chapter discuss these facilities in more detail, and are followed by a brief description of other operating systems available to 1900 Series users.

Job descriptions

Under GEORGE 3, any of the operating instructions which would normally be handed to the operator in a basic operational environment can be given to GEORGE instead, in the form of a *job description*. A job description is a set of instructions, called commands, similar to a program, written in a special *command language* which resembles simple English for ease of use.

Like programs, job descriptions are written on coding sheets and subsequently punched on cards or paper tape, which are read by GEORGE from the appropriate input peripheral. The job description is then stored in a *job description file*.

At the beginning of his job description, the programmer specifies one of two courses of action to GEORGE: either the programs described in the job description may be run at once, or the job description may be filed on backing store for later use. Jobs which run programs in this manner, without programmer intervention in the course of the run, are called *background jobs* (see Chapter 2).

Alternatively, instructions in command language may be input from a MOP terminal.

With MOP, the programmer types in his instructions from a typewriter console and GEORGE implements each one before inviting the programmer to type the next. In this situation the user in effect takes over the function of a job description file; however, it is possible to initiate a background job from a MOP terminal.

Since the job description is executed by GEORGE the operator's need to intervene constantly in the running of a program is clearly reduced. The resultant gain in efficiency is one of the major advantages which an operating system offers in a large installation where many programs are to be run simultaneously.

Internal management of central processor and core store

The way in which an operating system handles the internal management of the central processor and core store is a crucial factor in decreasing the turnround time of a job and increasing the installation's computing power. Multiprogramming facilities are provided by Executive to give more efficient use of central processor time, but the number of programs that can be run simultaneously is limited by the size of the core store and the number of available peripherals.

Under GEORGE 3, the number of programs that can be active at any one time is not restricted to the number that can be held simultaneously in core store, programs are held on fast magnetic media and brought into core whenever processing time is available. This process of *swapping* programs in and out of core store is carried out automatically by GEORGE. Furthermore, the off-lining facilities, described on page 32, remove the restriction on multiprogramming imposed by the number of peripherals on the installation.

The GEORGE 3 scheduling system decides what proportion of the machine's resources to allocate to each program on the basis of scheduling information supplied by the user.

The GEORGE 3 scheduling system is more fully described in Chapter 6.

Remote batch processing

A computer installation often consists of a central machine room, containing the required hardware configuration, with programmers at various locations remote from the machine room writing programs and sending them to the machine room to be run. For some smaller installations, this arrangement may not lead to any serious inconvenience as the programmers' location might be close to the machine room, so that transportation problems are not involved, and the volume of work being processed in the machine room is manageable. Many organizations, however, have branches or departments located at a considerable distance from each other and the existence of a single central machine room, besides being geographically inconvenient, can cause a serious bottleneck.

On-line programming is one method of giving programmers at remote locations direct access to a computer installation. However, the input and output facilities available with an on-line programming terminal are limited and do not allow for large scale input/output. GEORGE 3 therefore includes a remote batch processing system that allows users to input data and programs, and to receive output, on peripherals remote from the central machine room. This is done by means of 7020 terminals.

With the use of remote batch processing, job descriptions, programs and data are input from a basic peripheral at a location remote from the machine room. This input is read into backing store and programmers can then use it in the normal manner. Any output from the program may be routed by GEORGE to peripherals at the remote installation where the job originated, or at another remote installation. Remote batch processing therefore enables a small computer installation to be set up at a distance from the central processor and backing store.

The GEORGE 3 remote batch processing facility with MOP thus offers a great increase in efficiency to a user with a large installation whose facilities must be made available at widely separate locations.

Backing store and off-lining facilities

BACKING STORE

Under GEORGE, backing store is organized so that as much as possible of the data to be handled can be accessed by the central processor without intervention by the operator. Data on backing store can be held in one of two ways:

1 *In the filestore*. The filestore is the most important part of backing store under GEORGE 3. It is held on direct access storage devices (fixed and exchangeable discs and drums) and also on magnetic tape. Information in the filestore is logically organized into *files* which are *device-independent*; that is, they are referred to by name rather than by hardware addresses, so that the user need not know where his data is physically located at a given moment. GEORGE alters the physical arrangement of files within the filestore according to the requirements of the central processor. For example, a file may be read in from a pack of cards onto a disc or drum in the filestore. Later it will be dumped to magnetic tape as a precaution against its loss in the event of machine failure. The file can subsequently be moved from one device to another any number of times without the user's knowledge. The filestore enables users to store within the system all the information necessary for running a job. Since GEORGE monitors the input and output activities of the programs under its control, it can be made to direct the appropriate input and output action to device-independent files. Consequently programs need not be specially written to exploit the filestore facility.

   GEORGE protects users' files against unauthorised interference from other users by means of *user traps*. Each user file has one or more traps associated with it, each containing a user's name and listing the *modes* (for example, READ, and WRITE) in which the user is allowed to access the file in question. A user with READ access to a file can read from it but not write to it; with WRITE access a user can write to a file but not append to it, and so forth. A dumping system ensures that a minimum of data is lost in the event of machine failure.

2 *Outside the filestore*. The user may sometimes want information to be held in *device-dependent* form outside the filestore. This may be the case when he wishes to retain information on a tape which is to be used at another installation, for example, or on a tape to be used when the system is not running under GEORGE. Such files may be on either direct access devices or magnetic tapes, but only tapes can come under the GEORGE security system described above. Files thus stored remain device-dependent, and will not be dumped as a protection against machine failure.

OFF-LINING

In a normal environment, data transfers take place between core store and *basic peripherals*, that is card readers and punches, paper tape readers and punches, and line printers. Two factors reduce the efficiency of this system:

1 The slowness of basic peripherals compared to the high speed at which the central processor operates means that much of a program's run time may be spent transferring data to and from basic peripherals. During such

transfers the central processor is idle. Multiprogramming helps to compensate for these delays but does not completely remove them.

2 Multiprogramming is limited by the number of basic peripherals available for inputting and outputting programs at run time.

An Executive environment offers a partial solution to the problem by providing standard utilities to copy data to and from tapes, discs, and drums. Since magnetic media are relatively fast, the time spent in peripheral transfers during a program run is reduced. However, two disadvantages remain:

1 Processing time is taken up in running a standard utility before each run of the user program, and after each run if basic peripheral output is required.

2 Data transfer instructions in the user program must be rewritten to refer to the appropriate magnetic media.

GEORGE 3 provides a more sophisticated solution to the problem of data transfers. This is the *off-lining* facility.

Under GEORGE, data fed into the computer following an INPUT command is automatically copied to magnetic media in backing store. The format of the original file is retained and peripheral instructions can be automatically referred to the correct device by means of an ASSIGN command, so that programs written for basic peripheral input and output need not be rewritten. Output may be directed, again by means of an ASSIGN command in the job description, to a file in the filestore rather than to an on-line peripheral. Apart from reducing the time spent outputting data to a basic peripheral, this has the advantage that data thus held in the filestore may be accessed by any other program. Data may be output to a basic peripheral at any time, by means of a LISTFILE command. Off-lining also means that the availability of basic peripherals at run time does not limit the number of programs which could otherwise be run simultaneously under GEORGE.

MOP (Multiple On-line Programming)

The idea of *on-line* work is that a programmer has direct access to a central processor. He does not have to send his program to be punched, then submitted to an operator at an installation possibly distant from his own location; this eliminates the possibility of human error and delay at several stages in the proceedings and also cuts down on the loss of time involved in transporting the program to the machine room.

Obviously, if the programmer can bypass the operator, feed his program directly into the central processor and see the output immediately, program development is made far more convenient and efficient.

GEORGE's MOP (Multiple On-line Processing) facility permits a number of users to be on-line simultaneously to a central processor, while batch work goes on in the background. All GEORGE facilities relating to jobs and the filestore are available to MOP users. (See the appropriate chapters for more information.)

A convenient device for on-line work is a console typewriter, which is suited to the input and output of small amounts of information. The current version of MOP is designed for 7071 console typewriters. Later versions will cater for other devices.

MOP enables users to feed information into the system, to run programs, and to receive output. One or more basic peripherals may be simulated at the MOP terminal. This mode of operation is particularly well suited to jobs that fall naturally into small sections, between which human intervention is required: for example program development, inter-related file enquiries or editing. However, it is also possible to use a MOP terminal to initiate batch jobs to be run as background activities.

MOP is more fully described in Chapter 5.

Work scheduling

In a 1900 Series installation not running under GEORGE, the operator shares the task of *scheduling* with Executive; that is, when multiprogramming, the operator loads programs into the available core store and Executive allots central processor time to the program with the highest priority which is able to run at a given moment. When one program is suspended, for example, while awaiting a peripheral transfer, Executive scans the other waiting programs and runs the one with the highest priority able to run.

When a program is completed, the operator is informed and he loads a new program into the newly freed area of core store.

There are several reasons why this system is an inefficient way to manage the resources of the core store and central processor:

1 The need for physical intervention on the part of the operator in loading new programs results in considerable loss of time considering the speed of the central processor.

2 The operator must decide which program to load at any one time. This is a complex task, as he must take into account not only the available core store but also such factors as the peripheral requirements of the various programs; consequently it is likely that job mixes will not be the most efficient possible.

3 The system of Executive priorities is inflexible. The operator cannot ensure that a low priority job gets sufficient use of central processor, without constantly changing priorities. This difficulty becomes serious when on-line working, requiring a fast response to each user, is involved.

THE HIGH AND LOW LEVEL SCHEDULERS

The two GEORGE routines that control scheduling are the high and low level schedulers. The high level scheduler replaces the operator's function of selecting which waiting job is to be run, on the basis of requirements specified by users as well as peripheral requirements and other criteria. Jobs selected by the high level scheduler are passed to the low level scheduler, which shares out central processor time in accordance with policies determined by the high level scheduler. The low level scheduler is geared towards giving a reasonable response time to MOP users, although heavy demands necessarily create delays. These routines are described in Chapter 6.

Logging and accounting

In a typical installation, time on the machine will be bought by individual departments within the organization. Each job, or program, will be given an accounting number and the operator will keep a log of who uses the machine. At a later stage, these figures will be used to calculate the amount each department is to be charged.

Even in the most efficient installation, however, these figures cannot be completely accurate. It is difficult to estimate the time taken to run one particular program, particularly in a multiprogramming environment, and it is difficult to decide how much of that time was due to operator error or organizational inefficiency.

GEORGE 3 provides logging and accounting facilities to calculate precisely the amount of time used by each program and enables the installation manager to control the users of his installation by allocating to each of them budgets for resources such as mill time and magnetic tapes. GEORGE checks that users do not exceed these budgets.

As well as these built-in facilities, GEORGE 3 installations are also supplied with a log analysis program which will calculate accounts for each user of the system. This program can be modified by the installation manager to provide the kind of accounting information most appropriate to the installation.

The accounting and budgeting facilities are described in more detail in Chapter 7.

GEORGE 4

GEORGE 4 provides the same facilities as GEORGE 3, but is designed for use with a *paged* 1900. In these machines, the direct access backing store and core store of an object program are considered to be a unit, the *virtual store*, which is divided into *pages* of 1K words in length. These pages may be distributed throughout those parts of backing store and core store reserved for object programs, and only the particular page which is required need be in core store at a given moment. GEORGE automatically handles the swapping of pages in and out of backing store. Advantages of this system are that it is not necessary to find a contiguous area of core store large enough to contain the whole object program, and that backing store transfers are reduced, since only the required pages need be swapped into core store. It is also possible for two or more programs to have access to the same page in store. A page to be accessed in this manner must be written in *pure code*, that is, code which does not modify itself in the course of processing, since only one copy of the page is kept in store. GEORGE 4 uses this hardware facility to provide *shared programs*: that is, the same (pure) program operating simultaneously on two or more different data areas. (In an unpaged system, if more than one program has access to a section of coding, a separate copy will be kept in store for each program; consequently it does not matter if modifications in the coding occur.)

OTHER ICL OPERATING SYSTEMS

In addition to GEORGE 3 and 4, ICL supplies operating systems that work in conjunction with the normal Executive. These programs usually have some kind of privileged status which gives them facilities not available to other object programs. For example, an operating system might perform the extracode functions normally undertaken by Executive. Two of the ICL GEORGE operating systems, GEORGE 1 and 2, are of this type.

When multiprogramming, all programs are under Executive control and each can be regarded as occupying a *channel*. The number of channels available with a particular Executive is the maximum number of programs that can be multiprogrammed under that Executive. (A channel has no physical representation. It is simply a convenient way of describing the method by which Executive controls multiprogramming.) GEORGE 1 thus occupies one channel; however, it has the privilege of setting up within itself a user's object program, the PUC

(Program Under Control). Programs to be run under GEORGE are batched together into *jobs*; all the information necessary to run the job is input to the system as a *job description*, in the GEORGE 1 and 2 command language (which is similar to the one used with GEORGE 3 and 4) and where Executive would normally request operator intervention, GEORGE interprets the appropriate command as specified in the job description.

GEORGE 2, which consists of three programs occupying three channels, provides all the facilities of GEORGE 1 and, in addition, permits off-lining; two of its three programs are concerned with input and output between backing store and basic peripherals. Unlike GEORGE 3 and 4, off-line files may not be retained permanently in backing store, but are output immediately to slow peripherals. However, GEORGE 2 off-lining approaches that available with GEORGE 3 in that it is not necessary to rewrite programs to take magnetic media into account.

It is not necessary that all channels be occupied by GEORGE; some channels may contain programs under Executive, while others contain a copy of GEORGE and its PUC.

Other ICL software providing certain operating system facilities includes:

(a) Executive, as already described.

(b) Facilities supplied as part of the basic operational environment but which fulfil the operating system function of reducing operator intervention. The PLAN DELTY instruction is an example.

(c) Batch compilers which allow a sequence of programs to be compiled with little operator intervention.

(d) MINIMOP 1 and 2, providing a limited range of on-line facilities for the small user.

(f) Automatic Operator, which replaces messages from the typewriter console by pre-punched messages.

The remainder of this manual deals with GEORGE 3. Unless otherwise stated, references to GEORGE are therefore to GEORGE 3.

Chapter 2 Jobs

The unit of work submitted to GEORGE is called a *job*. The information necessary to run the job is input in the form of a *job description*, comprising a series of commands from the GEORGE command language.

COMMAND LANGUAGE

The language used to write job descriptions for jobs to be run under GEORGE is called the GEORGE command language.

Format of commands

A job description is written as a series of *commands*. Each command consists of an optional *label*, preceding a *verb* which may be followed by one or more *parameters*.

LABELS

A label is given if it is required to GO TO the command in question from some other command in a job description or macro definition file. The label serves to identify the command which is to be branched to and obeyed.

VERBS

A verb defines the operation to be carried out. LOAD, RUNJOB and WHENEVER are examples of verbs.

PARAMETERS

Parameters define the manner in which the command is to act. In the command specification sections of GEORGE manuals, all possible parameters for each command are given. Parameters are either mandatory or optional. If mandatory, they must be given each time the command is issued. If optional, they may either be given (in permitted groupings only, as defined in the command definition) or not specified. In some commands the parameter sequence is fixed; in this case, if an optional parameter is omitted between two parameters which are specified, the absent parameter must be indicated by a second comma following the first parameter.

In the EDIT command, for example, the parameter sequence is fixed; parameter A (*oldfile*) is mandatory, while B and C (*newfile* and *editfile*) are optional. Thus

```
EDIT    FRED
```

is permissible; so are

```
EDIT    FRED,
```

and

```
EDIT    FRED,,HARRY
```

In the first example, the second and third parameter are omitted; in the second, the second parameter is null (as indicated by the presence of the separating comma) while the third is omitted; in the third case the second parameter is again null while the third is specified.

Built-in commands

A *built-in* command activates a relatively simple GEORGE function, for example, loading or entering a program:

LOAD *filename*

ENTER

Built-in commands are implemented by built-in parts of the GEORGE program.

Macro commands

Job descriptions are usually a mixture of built-in commands and *macro* commands. The latter are used to call in more complex functions, such as compilers or the programs known to the system. A macro command expands into

one or more commands according to a definition stored in a file. The command as issued is the name of the file containing the definition.

A macro command may be issued followed by one or more parameters, which represent the values of the variables in the definition (see *Parameters in user macros,* below, for more detail on parameters).

A macro command may be either supplied as part of the system or defined by the user. In either case, macro commands are one of GEORGE's most important features. The macro concept means that, while the built-in facilities of the system are relatively limited, the system may be considered by the user for practical purposes to be very much larger, since built-in commands are supplemented by system or user macros which are used in exactly the same way. As more software is developed, it can be incorporated into the system, either as standard system macros supplied by the manufacturer or as user macros available to users at a particular installation; thus each installation can define its own command set in accordance with its users' requirements.

System macros

System macros are used to call in standard software produced by ICL for the 1900 Series, such as compilers. The macros FORTRAN, followed by the appropriate parameters, will activate the FORTRAN compiler. When GEORGE encounters a FORTRAN macro in a job description it finds the appropriate system file and expands the macro into the commands necessary for loading the FORTRAN compiler from backing store, entering it and running it.

A system macro such as FORTRAN and a built-in command like LOAD are issued in exactly the same way.

Note: System macros like FORTRAN which are described in this manual may be superseded by new macros. System macros will be defined in the relevant software manuals, and users should refer to these manuals to be sure of having up-to-date information on any system macros they wish to use.

User macros

If a user wishes to repeat a sequence of commands a number of times, he can define his own macro command and store it in a file. Later the whole sequence can be implemented by calling the macro with the necessary parameters.

For example, a user might define a macro RUNPROG which expands into the commands

LINGO *sourcefile, objectfile*
LOAD *file description*
ENTER *number*

Note: LINGO is a fictional system macro, analogous to FORTRAN which loads and runs the FORTRAN compiler. LINGO has been substituted where examples of a system macro are required in this chapter, since specifications of actual system macros are subject to alteration.

Command processor levels

When a job is initiated by a JOB or RUNJOB command, the source of commands to GEORGE changes from the card or paper tape reader (or MOP terminal) to a temporary or permanent job description file. The JOB command is normally issued from a basic peripheral, but the commands that follow JOB and constitute the job description are stored in a file and subsequently issued from it. This change of the source of commands is called a change of *command processor level.* The JOB command is said to be obeyed at command processor level zero; the commands in the job description file (LOAD, ASSIGN, ENTER, etc.) are obeyed at the next lower level, in other words at command processor level one.

MACRO LEVELS

Within a given job the command processor level is increased if a macro in the job description file issues one or more commands. For example, if the job description contains a macro, the commands within the macro definition file will be obeyed at a command processor level one lower than that at which the macro is issued. If the macro is issued at level one, the commands that constitute the macro will be obeyed at level two. If one of the commands within this macro definition file is itself a macro (called a *nested* macro), it will cause a further increase in the command processor level. Every time the source of commands changes to a new macro definition file, the command processor level increases by one. When the program exits from the macro control will be returned to the level one higher than that at which the macro was obeyed.

Parameters in user macros

In the RUNPROG user macro defined earlier in this section, the problem of assigning parameter values arises. An ordinary job description to compile the FORTRAN source program contained in file named JACK, place it in file JOE, and enter it at entry point 1, would be written:

LINGO JACK,JOE

LOAD JOE

ENTER 1

If this sequence of commands is defined as a macro RUNPROG, as above, the parameters are included in the original macro definition and the commands would be issued simply as

RUNPROG

THE PARAMETER BLOCK

When a user macro is issued it has made available to it a *parameter block* of twenty-four variable length *parameter locations;* these are named A to X in alphabetic sequence.

PARAMETER IDENTIFIERS

Many commands are followed by a string of parameters, which are specified when the command is issued. In the case of commands within a macro, actual parameter values may be replaced by a string of *parameter identifiers* if it is intended that the macro be used with a series of different values. Each macro may have a total of up to twenty-four of these, from A to X, each corresponding to the parameter block location identified by the same letter; actual values are stored in the parameter block locations. When a macro is executed, GEORGE substitutes the actual values for the given parameter identifiers.

The execution of the RUNPROG macro requires three parameters: source filename (actual value JACK), object filename (actual value JOE, later the name of the file to be LOADed) and the number indicating the entry point (actual value 1). For the purposes of this example JACK will be stored in location A of the parameter block, JOE in location B, and 1 in location C. When RUNPROG is expanded, parameter identifiers will therefore be as follows:

LINGO %A, %B

LOAD %B

ENTER %C

The macro will be called as RUNPROG JACK,JOE, 1; at run time, GEORGE will substitute for %A the value held in location A, that is, JACK; for %B, JOE; for %C, 1.

THE SETPARAM COMMAND

If the user wishes to repeat the macro using different data, he must re-issue the macro call specifying new parameter values. However, under some circumstances the user may wish to alter parameter values in the course of a run.

The SETPARAM command may be issued from within the macro whilst the macro is running to allow values held in the current parameter block to be reset.

The use of the SETPARAM command is fully dealt with in the GEORGE 3 and 4 manual

Conditional commands

Conditional commands are an important feature of the job description. There are two of these built-in commands, IF and WHENEVER.

The IF command enables the user to specify alternative courses of action according to whether or not certain conditions have been satisfied. Typical examples of such conditions are program halts; messages; failures; states of switches. The format of the IF command is shown in the example:

IF FAILED, GO TO 1A

where the command (GO TO 1A) which follows the condition (IF FAILED) indicates the course of action to be taken if the condition is satisfied.

With the WHENEVER command, the user specifies the action to be taken whenever certain categories of event occur:

WHENEVER BREAKIN, GO TO 2

WHENEVER COMMAND ERROR, GO TO 1A

.

.

.

LOAD JACK

.

.

.

ENTER

IF FAIL, GO TO 1A

The WHENEVER command applies to all the commands following it, until a new WHENEVER command specifying the same event is issued, at which point the original command is superseded. If the condition specified in the first parameter is encountered, the WHENEVER command is unset before the command given in the second parameter is obeyed. The events which permit a WHENEVER command are: command error, break-in and finish.

These two commands supply the decision-making function of the programmer working on-line, or of the operator working in a non-GEORGE system.

Program event messages

When a program event occurs, a *program event message* is written to a special area of core store. A job can have only one program event message at any one time, that is each program event message overwrites the previous one. The messages associated with HALTED and DELETED events consist of the messages generated by the extracodes that cause the program events. For FAILED events, the message gives details of the failure.

To define the condition of an IF command more strictly, the user can include in the command a character string to be compared with the program event message. The character string must be enclosed in parentheses or quotes; if spaces are to be significant in the comparison, quotes must be used.

The user need not give as his character string the entire text of the program event message. The condition will be satisfied if the *n* characters specified in the IF command agree with the first *n* characters of the program event message.

Program event messages are also written to the job's *monitoring file* (see below).

Types of job description

PERMANENTLY STORED JOB DESCRIPTIONS

In the standard type of background job, all information necessary for the running of the job is stored in permanent files in the filestore. Programs are loaded from the filestore, and all peripheral transfers are off-line transfers between core and filestore. A job with a permanently stored job description is initiated by the RUNJOB command.

ONCE-ONLY JOB DESCRIPTIONS

When a job description is to be used only once, it may be stored in a *working job description file* which will be erased as soon as the job is terminated. A once-only job description is introduced to the filestore, and the job initiated, by the JOB command (see Chapter 12 of *Operating Systems GEORGE 3 and 4* for details of JOB and RUNJOB.

Monitoring files

Each time a job is begun, GEORGE creates a *monitoring file.* This file holds information which would be sent to the operator's console both from Executive and from the programs comprising the job in a non-GEORGE environment. This includes the output of program extracodes and logging information. Some categories of monitoring file data are also output on the operator's console or MOP terminal. The user may specify which categories of information are to be stored in the monitoring file by means of the TRACE command, and also those which he wants output. These categories are described in full in the GEORGE 3 and 4 manual.

When a job is terminated, by an ENDJOB or LOGOUT command, its monitoring file is closed and GEORGE uses its contents to calculate the charge for running the job (see the GEORGE 3 and 4 manual).

By means of parameters in the terminating command of the job, the user can indicate which categories of monitoring information he wishes to have listed. In this way the operating system can assist in the task of analysing a program run.

INPUT/OUTPUT FOR OBJECT PROGRAMS

Input/output operations under GEORGE may be either off-line or on-line. Normally most work will involve the off-lining facilities.

Input and output using off-lining facilities

In a typical GEORGE job, data is input to permanent files in the filestore from a basic peripheral. Output is initially stored in the filestore and eventually transferred to a basic peripheral.

INPUT TO FILESTORE

Data is transferred from a basic peripheral to the filestore by means of the INPUT command. This command creates a file in the filestore containing all the data following the INPUT command up to a terminator.

The INPUT command inputs data separately before a program is loaded or run. It is possible to include data in a job description by means of an *embedded* INPUT command which initiates the transfer only when the job is begun.

Embedded data, not preceded by an INPUT command, may also be included in the job description file and read directly from it by a program previously loaded in the job description.

INPUT/OUTPUT VIA SIMULATED BASIC PERIPHERALS

The ASSIGN command opens a filestore file associated with a peripheral channel of an object program, so that control instructions for the peripheral are referred to the filestore file. As a result, programs originally written for basic peripherals need not be rewritten to be run under GEORGE.

OUTPUT VIA BASIC PERIPHERALS

The LISTFILE command causes some or all of the data in a file created by an ASSIGN to be output off-line via a basic peripheral or the monitoring file.

For a more detailed description of these input/output facilities, see chapter 4.

Input and output using on-line peripherals

Basic peripherals can be connected on-line to a GEORGE-controlled program in the same way as they can to a program running under Executive. Under GEORGE, the ONLINE command, issued in the job description, makes the peripheral available to the program.

This command is further discussed in Chapter 4.

TYPES OF JOB

Jobs run under GEORGE are of two types: *on-line* jobs and *background* jobs.

On-line jobs

The most significant feature of an on-line job is that the user is informed at every stage of what is happening to his job and can act accordingly. This is possible because information sent to the monitoring file is output on a typewriter console as the job progresses. If, for example, a program fails to compile, the programmer will be informed of the failure as soon as it occurs and can correct the error in the source program immediately by performing an edit (see Chapter 5); he can then attempt the compilation again and, if it is successful, run his program. This facility is advantageous when the program's requirements cannot be met by using the conditional commands (IF and WHENEVER, see page 9). In this case, the user is communicating with GEORGE; it is also possible to communicate with a program under GEORGE's control.

The on-line user can also input data, including program and job descriptions, from his terminal, and have output directed to it.

The GEORGE on-line facility is called MOP (Multiple On-line Programming). MOP is described in more detail in Chapter 5.

Background jobs

Background jobs differ from MOP jobs in that monitoring information is not output for the user's immediate scrutiny. Consequently, programs to be included in a background job should come within the decision-making scope of the conditional commands included in a previously written job description.

A background job may, however, be initiated from a MOP terminal, or it may be a MOP job which has been disconnected because it no longer requires intervention by the programmer. This may be the case with programs which have been tested on-line and are to be run with input data already in the filestore. Jobs can also be disconnected and allowed to run independent of the MOP terminal if the user has issued a command which will take some time to process. While the job is disconnected, the user can do other on-line work and later, if he wishes, either reconnect to the original job or connect another background job to the MOP terminal.

4265(4.71)

Chapter 3 Backing store

THE ENTRANT CONCEPT

Data held on backing store in a GEORGE environment is contained in *entrants*. These may be inside or outside the GEORGE filestore; entrants inside the filestore are known as *files*, whereas entrants outside are either magnetic tapes or *exofiles* (direct access entrants outside the filestore).

The word 'entrant' is used because, in the context of GEORGE, the word 'file' has the restricted meaning of 'filestore file'. The significance of the word 'entrant' will become clear when directory entries are explained (see page 15).

The contents of an entrant

An entrant may contain any finite, ordered sequence of words or characters. It may consist of a user's program in any language or in the form of a binary dump, an item of standard 1900 Series software, a batch of data for a user's program, or data produced or required by GEORGE. In short, almost all the information involved in running jobs under GEORGE may be held in entrants.

Most of this chapter is concerned with files (that is, filestore files) since these normally comprise the bulk of backing store at an installation using GEORGE. Other kinds of entrant and their relation to the filestore are explained on page 27.

THE FILESTORE

Device-independence

Unlike conventional files, GEORGE files are not held on any one particular storage medium. They may be distributed over all the available backing store, on both direct access devices and magnetic tapes. The user does not have to know where a particular file is held at a particular time, since he identifies files by their names, not by their hardware addresses. Thus, if occasion demands, GEORGE can alter the physical arrangement of the files within the filestore without the user's knowledge. A file that the user treats as a continuous series of blocks may in fact consist of fragmented elements scattered throughout the filestore.

Types of file

In the current mark there are three main types of file within the filestore, *basic peripheral files, direct access files* and *magnetic tape files.* If a user wants a file to simulate a basic peripheral, so that programs can handle the file serially in the same way as they handle basic peripheral data, he must create a basic peripheral file. If the user wishes to read from or write to specific addresses within the file, he must create a direct access file. Magnetic tape files simulate magnetic tapes in the filestore. The user will often choose the type of file in accordance with the specifications of the software he is using; when he has a free choice, basic peripheral files are usually the most convenient. In all cases the size of a file is limited to 245K words. However there is a *multifile* facility which effectively allows larger files to be used.

BASIC PERIPHERAL FILES

Although all basic peripheral files consist of strings of records, their formats vary. One reason for this is that their initial input medium may have been cards or paper tape, and in most cases the format of a file must correspond to its input medium. A second reason is that, regardless of the input medium, the format of a file must be compatible with the transfer instructions in the program to which it is connected as the source or destination of data. This does not mean that, for example, a file in card reader format (a CR file) can be processed only by PERI instructions for a card reader; the format of a CR file is also compatible with card punch PERIs. Similarly, though CP and LP files are normally written to by card punch and line printer PERIs respectively, they may also be read by card reader PERIs because their formats are compatible with this type of transfer instruction.

A user's basic peripheral file has the format appropriate to the peripheral it simulates so it may be a CR, CP, LP, TR or TP file. The user, however, has greater scope in handling a basic peripheral file than in handling the corresponding

device. For instance, the user can arrange for a program to write to an LP file with line printer PERIs and can then use this file as the source of input for a second program. He does not have to transfer the output data to cards or paper tape and load it on a basic peripheral, since LP files can be read by card reader PERI instructions.

If the user should require a basic peripheral file longer than 245K words he may use a *multifile.* A multifile is a set of files with the same format and the same name, except that the generation number is increased by one for each file. The file with generation number one is used as an index to the other files which are called the *elements* of the multifile. This facility is a temporary one which will be replaced later by a comprehensive facility hidden from the user.

It is intended that eventually all operations using basic peripheral files should be able to operate on any basic peripheral file. Basic peripheral files are held by GEORGE in several different formats. In certain cases where a file is read by a PERI of a different type or mode to that with which the file was written, it becomes necessary to convert the file to the correct format to be dealt with by the reading PERI. However, this drawback does not apply to simple cases such as the off-lining of cards to be read by a program; such cases are dealt with more efficiently than they would be if all files were held in a single internal format and each file had to be converted each time. There are certain restrictions to the present system that the user needs to be aware of; some categories of conversion are not possible at present, some conversions are slow and should be avoided if possible, and some conversions necessarily cause information to be lost. For example, information is lost if a paper tape has been read into a filestore file in NORMAL mode and the file is subsequently read by a program in ALLCHAR mode. Details of the different internal file formats may be found in the manual *Operating Systems GEORGE 3 and 4* and details of which conversions are possible in the current mark are given in the same manual.

System files and directories (see below) are usually also handled serially, but they do not correspond in format to basic peripheral files and are therefore known as *amorphous* files.

DIRECT ACCESS FILES

A direct access file must have either drum or disc format; there is no difference between an ED file and an FD file. Because of the different ways in which disc files and drum files (DR files) are addressed, it is not possible to use PERIs for disc to access a drum file, or vice versa.

It should be noted, however, that this restriction derives not from the medium on which the file is held, but from the way in which the file is organized. A disc file may be held on drum (or vice versa), but, regardless of its storage medium, it will be organized as though it was held on disc.

MAGNETIC TAPE FILES

A magnetic tape file must have a magnetic tape format. There is a difference in that the first MT block of the file, which corresponds to a tape's header label record, need not have a header label format. The details in the control area of an open PERI to this tape file will not be checked against the contents of the first MT block; thus it is not necessary for this MT block to have the standard header label format. In other respects, since the file will be written to and read from by standard magnetic tape data transfer instructions, the format of the file will conform to magnetic tape standards.

Modes of access

User programs are connected to files by means of the ASSIGN command. This command causes a named file to be opened and associated with the current program in such a way that PERI instructions for a specified peripheral are simulated by reading from or writing to the file.

If the file is a basic peripheral file, the only possible mode of access is *serial access.* Serial access in the input mode is the reading of each record in the file, starting at the first record, going on to the second record, and so on. Serial access in the output mode is the writing of successive records, either starting at the beginning of the file and going on to the end, or starting at the end of the file and writing successive records up to a new end of the file. Writing records to a serial file is generally referred to as *appending.* The term 'append' has a more restricted sense in the context of user traps (see *Privacy,* page 19).

It is possible for a serial file to be accessed by more than one program peripheral channel simultaneously. Any number of channels can read from the same serial file at the same time and in the current mark a file may be shared by peripheral channels appending to it and reading it, subject to certain controls. A serial file that is being read by one or more peripheral channels whilst others are appending to it is known as a *communications file.* GEORGE must be informed if a file is to be used in this way. It will then ensure that the file is in a basic peripheral format, so that the reading and appending programs can access the file simultaneously in a serial mode. Briefly, GEORGE will organize the processing of communication files in the following way: if an appending program is ready to read

the same record, the information will be passed across in the core store, thus eliminating a backing store transfer (the record will still eventually be written to backing store, but it will have to be read again); if the reading program attempts to read a record that has not yet been generated by the appending program, the reading program will, unless direct response mode is being used (see Chapter 14 of *Operating Systems GEORGE 3 and 4*), automatically be suspended and reactivated when the record becomes available.

Direct access files are accessed randomly. This means that the user program PERI instruction specifies an address within the file where reading or writing is to start. Successive addresses that are specified need not be in any sequence. A direct access file cannot be a communication file, in the sense mentioned above, since it is not in general possible for GEORGE to organise the synchronisation of the programs accessing it. However, it is possible for one or more programs to read from the same direct access file at the same time as another program is writing to it.

Magnetic tape files are accessed in the same manner, and by the same instructions as a real magnetic tape. The file is always opened in such a way that the first non-open mode PERI will operate on the MT block after the first MT block in the file. This initial positioning procedure is compatible with GEORGE's handling of ONLINE standard tapes. More than one user may have a magnetic tape file open for reading at one time. COMMUNE writing (see the manual *Operating Systems GEORGE 3 and 4*) is not allowed to magnetic tape files.

The structure of the filestore

To enable users to refer to files by names, GEORGE keeps a *Dictionary*, held in the filestore, which consists of a number of entries, one for each user. Each entry contains, among other things, the name of a user's *proper directory*. A directory is a file that holds information about a particular user's files. These files may themselves be directories associated with users, or they may be *terminal* files, that is, files containing user information such as programs or data.

This arrangement gives the filestore a hierarchical structure. At the top of the hierarchy is a *master directory*, which contains information about files at the level below. If any of these files are directories, they contain information about files at the next level down, and so on to any depth. The effect is to give the filestore a tree-like structure, as in Figure 1 on page 16.

This diagram shows a very simple four-level filestore. The terminal files, or users' files, are represented by boxes containing names beginning 'TF'. Each user's file is directly linked to a directory above it indicated by a circle containing a name beginning 'DIR'. To show that each directory has a user associated with it, each circle has a user name outside it, beginning :USER (user names always start with a colon). This is explained under *Filenames* below.

Each directory in the filestore contains a number of directory entries, one for each of the files at the level below. In the case of the highest directory in Figure 1, the directory entries consist of information about the three directory files at the level below. The entries in these directories contain information about the directories and terminal files at the third level, and so on. The path leading from any file in the filestore to the master directory is unique. It may include several directories, and the file is said to be *inferior* to each of these. A file that has an entry in a directory is said to be *immediately inferior* to that directory and is regarded as *belonging* to the directory, and so to the user associated with it, who is said to be the *owner* of the file. Thus the terminal file TF2A, in Figure 1, belongs to the directory labelled DIR1A and is owned by user :USER1. File TF2A is immediately inferior to DIR1A and inferior to DIR0. The practical significance of this hierarchy of relationships will become clearer when filenames are explained below.

A directory, with user name :MANAGER, is issued to the user with the initial filestore (see Chapter 5 of the manual *GEORGE 3 and 4 Operation Management*). This directory is regarded as being at *depth* 0. The user can create further directories up to a maximum depth of 64 by using the MAKEDIR command; the filestore can therefore have a maximum of 66 levels, including terminal files (that is a terminal file at the 66th level will be at depth 65 since depths are numbered from zero, and will be contained in a directory at depth 64). In certain circumstances, however, GEORGE can create a temporary directory for a job running in a directory at depth 64. This temporary directory will be at depth 65, and its terminal files at depth 66, giving a maximum of 67 levels.

THE CONTENTS OF A DIRECTORY

Directories are serial files, and each entry consists of a group of records. The first record of each directory entry contains the local name of a file (see *Filenames*, below), information such as the file generation number, date and time written, details concerning the structure of the file (serial or direct access) and information needed by the back up and security systems. Next there are a number of records containing the addresses of the blocks occupied by the file, one record for each type of direct access backing store on which the file is currently held. When a file or part of a file is moved from one physical position in the filestore to another, these records are updated by

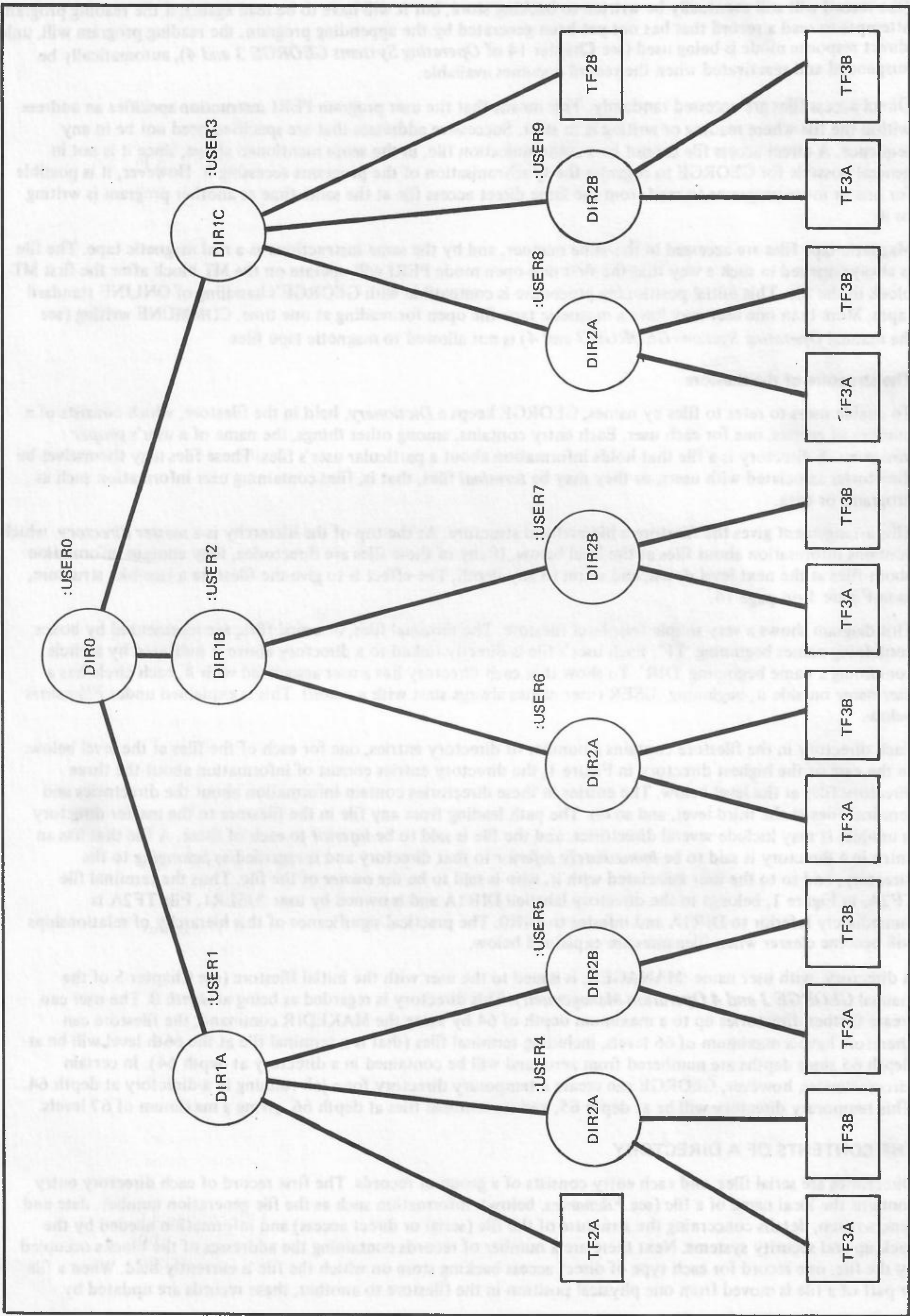


Figure 1 Filestore structure

4265(4.71)

the system. Finally there is a group of records which indicate which users are allowed to access the file and how they may access it.

Filenames

Since a user's directory indicates the names and physical addresses of all the files that belong to him, GEORGE can always locate a file that is referred to by name, provided that the name that is given enables GEORGE to find the relevant directory.

Each file, whether a directory or a terminal file, has a *local name;* for example, TF2A or DIR1A in Figure 1. It is not possible to have two files with the same general local name entered in the same directory, although it is possible for two files with the same name to be entered in different directories. Thus, in Figure 1, the directories DIR1A, DIR1B and DIR1C all contain entries for files called DIR2A and DIR2B. Since it is possible for the filestore to contain a number of directories with the same name, it is not necessarily possible to uniquely identify a file informing GEORGE simply of its local name and the local name of the directory in which it is entered (in Figure 1, page 16, there are three files called TF3A, entered in directories called DIR2A). For this reason each directory must have a second name associated with it, and this second name must be unique within the filestore. This name is the *user name,* for example :USER4, :USER6 or :USER8 in Figure 1. No two directories may have the same user name associated with them. This means that a user can uniquely identify a file by including in his file name an explicit or implicit reference to the user to whom the file belongs.

REFERRING TO FILES

A user who wishes to refer to a file must use a name that informs GEORGE of the logical position of the file relative to a user. To do this he may use either an *absolute name* or a *relative name.*

Absolute names

Any file in the filestore may be referred to by an absolute name. In the simplest case, where the user is referring to a directory, the absolute name may be just the user name associated with the directory. For example, to return to Figure 1, any user may refer to the directory DIR1A by the name

:USER1

This gives GEORGE enough information to find the physical address of the directory, with the aid of its Dictionary of users.

If a user wishes to refer to a terminal file by an absolute name, he must at least give the user name associated with the directory to which the file belongs, followed by the local name of the file being referred to. For example, any user can refer to the file TF2A by the name

:USER1.TF2A

(Component names in filenames are always separated by points.)

Alternatively, the user may begin an absolute name with the user name associated with some other superior directory. In this case the logical relationship between the named user and the file that is being referred to must be made explicit. This is done by including, after the user name and before the local name of the file, the local names of any lower superior directories (not the user names associated with these directories). These must be given in order of seniority, starting with the local name of the directory that belongs to the named user. For example, instead of using the name :USER1.TF2A, a user might refer to TF2A by the name

:USER0.DIR1A.TF2A

Relative names and the current directory

Associated with each job at any time there is a *current directory.* The user name associated with this directory is sometimes referred to as the *current user name.*

If a file is referred to by a name that consists of one or more local names, without a preceding user name, the filename is taken to be relative to the current directory of the job. In other words, if no user name is given, the current user name is assumed. Names of this kind are known as relative names.

The current user name is used only to simplify the naming of files by jobs. The user who initiated the job (for example by a JOB command) is called the *proper user,* or simply the user. It is the proper user who pays for the job and whose access to entrants in the filestore is controlled by GEORGE (see page 19, *Privacy and Accounting*).

At the start of a job the current directory is the same as the proper user's directory. However, the job's current directory may be changed any number of times by means of the DIRECTORY command. By using this command, the user can alter his base of reference, so that he can use relative names where previously absolute names were necessary. For example, if user :USER4 in Figure 1 starts a job, he can initially refer to only two files by relative names, TF3A and TF3B. To refer to any other file in the filestore he must use an absolute name. However, if he changes the job's current directory from DIR2A to DIR0 by means of a DIRECTORY command, he can use a relative name to refer to any file in the filestore apart from DIR0. Thus, he can refer to TF2B by the name

DIR1C.TF2B

instead of using an absolute name such as

:USER3.TF2B

or

:USER0.DIR1C.TF2B

Reasons for the structure of the filestore

It will probably be the case that all levels of management in an organization will need to control files of data, and different departments within an organization will need to keep their data separately. This is equally true of universities, public authorities and commercial concerns. There is, then, a correspondence between an organizational structure and the way the filestore has been designed.

The naming system of an organizational hierarchy will depend very much on what the organization is doing. To take the case of a fairly large company which purchases raw materials, processes them in a factory to produce finished goods and sells these goods to consumers, the company structure might be as in Figure 2, below.

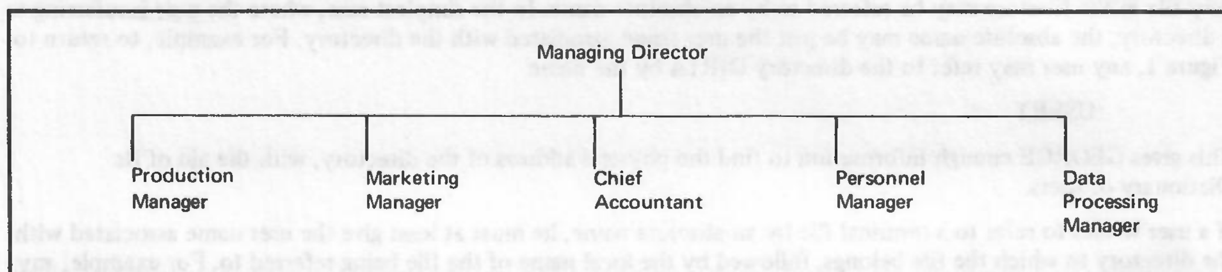


Figure 2 Company structure

Each of the line managers reports to the Managing Director, and each would typically control a number of staff, who would again be organized in a hierarchy. The department of the Data Processing Manager might be organized as in Figure 3.

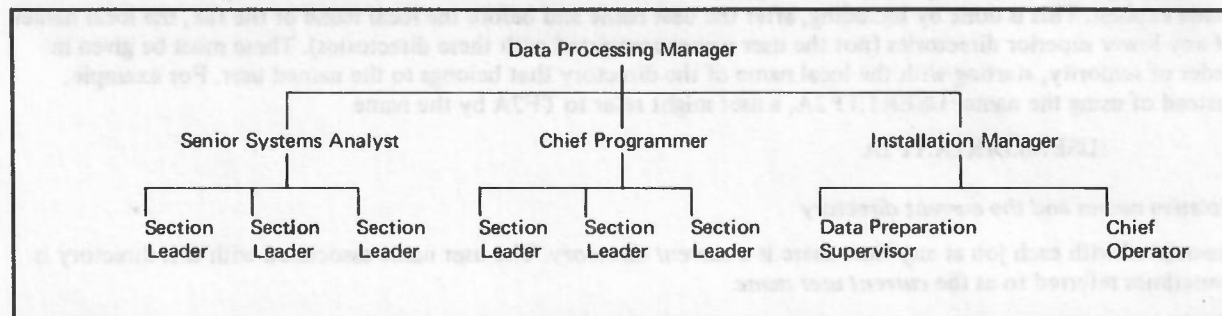


Figure 3 Company structure 2

The Senior Systems Analyst, the Chief Programmer and the Installation Manager would all report to the Data Processing Manager. Similarly, the Section Leaders under the Chief Programmer would report to him, and so on. There might be further levels below the lowest level of management shown.

The filestore of the hypothetical company in the example above can be made to reflect the company structure. For example the Data Processing Manager will have a user name and a directory. This directory will contain details about the three directories that belong to it and also about any terminal files that the Data Processing Manager may

4265(4.71)

create. Each user in the sytem may be either a single person or a group of people associated with a particular area of the firm's activities. For example, the user name associated with the Data Preparation directory need not be the name of the Data Preparation Supervisor. It may be a collective name for all members of this section who are allowed to use the Data Preparation files.

ACCOUNTING

One consequence of the tree-like structure of the filestore is that strict budgetary control can be exercised over the use of an organization's computing facilities. The senior user in the system, the Managing Director in the example, can share out the available computing facilities (time and space) among the users at the level below him. These in turn can distribute some or all of their share of these facilities among the users under their control, and so on down the hierarchy. GEORGE will ensure that users do not exceed their budget allocations and will, by means of charging algorithms, calculate what proportion of the running costs of the installation should be borne by each user.

In this way the role of the computer in an organization can be made subject to the normal rules of cost accountancy, and statistics vital to higher management, such as the rate of return on investment, can be conveniently calculated.

PRIVACY

One of the features of non-computerized data processing is that many copies of what is essentially the same file are held in different places. This is because the users of these files are physically remote from each other. Given the capacity of a computer installation to hold large volumes of data in an easily accessible form, and also the convenience of on-line working, the need for wasteful duplication of files disappears. Since the whole of a company's data can now be stored in one place, however, it becomes very important to specify who can and who cannot have access to information, and equally important what sort of access is permitted; for example a certain user might be allowed to read a file but never under any circumstances to alter the information in it.

GEORGE can be told to grant particular users access in specified modes to certain groups of files. Similarly GEORGE can be told that certain users have restricted access, or none at all, to other files. The Managing Director of a company, for example, might wish to ensure that nobody but himself ever had access to certain of his files. GEORGE thus makes information available to those who need it and minimizes the risk of unauthorized manipulation of data.

Unless a user (single or collective) takes deliberate action to grant other users access to some of his files, he alone is permitted to access them, and even then only in specific modes chosen by himself. This system of protection is implemented by means of *user traps.*

User traps are stored in each directory entry. Entry user trap contains the name of a user who is allowed access to the file concerned, and indicators that govern the modes in which access is permitted. The modes of access are READ, WRITE, EXECUTE (of files containing programs) and APPEND (that is, append to the end of a file). A user attempting to open a file in a particular mode will only be allowed to do so if there is a user trap that contains his name associated with the file, and if it has an indicator corresponding to the required mode, that is, only if the trap is open. If a user wishes to grant himself or another user access to one of his files in certain modes, he must inform GEORGE with the appropriate command (TRAPGO). A user may also issue a TRAPGO specifying a group of users instead of a single one. This causes a *group trap* to be set. Later, if he wishes, he may remove his permission by means of a TRAPSTOP command. The traps initially granted to the owner of a newly created file vary with the type of file. The owner of a basic peripheral file has READ, EXECUTE and APPEND access to it after it is closed (unless he elects to alter the traps); while it is still open he has WRITE access as well. The same applies to magnetic tape files and direct access files, except that the owner is not automatically granted APPEND access to either. Since a user can only control access to files that belong to him, these facilities afford complete protection against unwanted interference by other users and also against unintentional destruction by the owner At the same time they enable a user to refer to any files in the filestore as long as he has the cooperation of the owner.

THE BACK-UP SYSTEM

The back-up system in GEORGE 3 provides two facilities:

1. It enables GEORGE 3 to be started or restarted after a breakdown, with a recent and usable version of the filestore.
2. It enables the filestore to occupy more space than is available on direct access store, and helps prevent and clear backing store jams.

When GEORGE is loaded, it checks whether a valid version of the filestore is present on direct access backing store. If it finds that some directories or vital system files are incomplete, a *general restore* is initiated with the operator's assistance; in this case the existing filestore is ignored, and GEORGE restores enough of the filestore from the incremental dumps on magnetic tape to enable it to start running jobs. (Note that information created since the last dump will be lost.) Normally, however, a general restore will not be necessary. In either case, after GEORGE has started running jobs, access may be needed to files which are not on-line; these are restored as required from tape by the file retrieval system.

The *incremental dumper* is a special job which runs a program at regular intervals in order to preserve on magnetic tape copies of files which have recently been changed or created. Each time it is entered, the dumper searches the filestore for such files and writes them to tape as an *increment.* A copy of the complete filestore is thus held as a series of increments on the dump tapes. During dumping it is not possible to open files, and this may cause slight delay to jobs running at the time. As well as being run at regular intervals, the dumper may also be started by the operator or when the available backing store space is nearly full. In the latter case, files may be dumped to tape in order to allow backing store space to be freed.

Further details of the back-up system may be found in the manual *GEORGE 3 Operation Management*, but in general its operation does not concern the user. An exception is the RETRIEVE command which forms part of the file retrieval system.

The file retrieval system

From time to time, jobs will need files that are not on-line, and so have to be read from tape. All requests for such files are coordinated by the Retrieval System, which will put the requests on a queue, ordered according to the order of the files on the tapes, and open tapes appropriately. The user may make such requests implicitly by the ASSIGN, LISTFILE or LOAD commands, or explicitly by the RETRIEVE command, which puts requests for the named files on the queue and allows the user to proceed.

It is advantageous to use the RETRIEVE command in job descriptions unless it is certain that the file in question is in fact on-line. Usually it should appear near the beginning of a job description or macro, requesting all the files that will be used during the job. If the files are on-line already, there is no effect (although there is a small overhead) and if they are not they will be retrieved autonomously and according to their order on tape, which is clearly more efficient than retrieving them in the order in which they are mentioned in the job description.

The Juggernaut restorer

When a general restore has been done from the latest increment, it is possible that many of the most often used files in the filestore will no longer be on-line. Users may RETRIEVE files from the dump tapes. However, for greater efficiency, a job under :MANAGER, called the Juggernaut, is automatically run after a general restore and retrieves from dump tapes those files which the manager considers should be on-line when sufficient space is available. These files will only be dumped if they are infrequently used or if a particularly bad backing-store jam occurs. They will be brought on-line if the general restore does not include them.

Users need not attempt to RETRIEVE files which will automatically be restored by the Juggernaut. System files, also are automatically restored and should not be RETRIEVED.

The dump tape processor

This is an operator-initiated routine, which aims to keep the number of tapes containing incremental dumps to a reasonable figure. When entered it finds one of the tapes on which the oldest unprocessed reliable increment is dumped. It searches down the tape checking the directory entry of each file it encounters. There are three possible states for each file:

1. There is a later dump of the file, or no reference to the file in the filestore; in either case the file is ignored.
2. There is a copy of the file on backing store but no later dump. In this case the copy on backing store is marked 'to be dumped'.
3. There is no later dump or copy of the file on backing store. Here the file is restored to backing store and marked 'to be dumped'.

When a tape is found with files only in state 1 above, the increments on it are marked as obsolete and the dumper will release the tape after the next increment has been created.

THE EDITOR

The editor which is built in to GEORGE 3 enables the user to make detailed alterations to almost any basic peripheral file. The old file is edited into a new file according to editing instructions supplied by the user. These editing instructions index a particular character in the old file by reference to a *conceptual pointer* (see below).

Several old files may be edited to produce the new file; files can be of any peripheral type and the editor produces a new file of the same type as the first old file opened. The file containing the editing instructions and any new text to be inserted can also be of any basic peripheral type and need not be of the same type as the old file. Editing instructions and text can also be input from a MOP terminal.

Calling in the editor

To call in the editor, the user issues an EDIT command.

The format is:

EDIT *oldfile,newfile,editfile*

where

oldfile is the file description of the old file to be edited

newfile is the file description of the file to which the edited information is to be written

editfile is the file description of the source of the editing instructions.

The *oldfile* may be any basic peripheral file, other than a directory, to which the user has READ access, except for certain major system files. The *oldfile* parameter is mandatory and its omission will result in an error message.

The *newfile* may be an existing basic peripheral file to which the user has WRITE or WRITE and APPEND access. If the file described does not exist, the editor will create a file of the same peripheral type as the old file. If this parameter is null or omitted, the editor will create a file identical to the old file, except that the generation number will be increased by one.

The editing file or files must be basic peripheral files to which the user has READ access. If this parameter is null or omitted, the editing instructions will be taken from the job source: either the job description file or a MOP terminal. If the user is inputting from a MOP terminal, GEORGE will reply EDITOR IS READY, followed by an invitation to type, after the files specified in the EDIT command have been opened.

CONTEXTUAL RESTRICTION

The only context in which the EDIT command is forbidden is NO USER.

Editing language

The editing language is a series of *editing instructions.*

RECORDS

The editing file, like other files, is divided logically into *records.* Editing instructions are set out in lines, each line representing a record. One record may contain several instructions separated by commas. It is not normally possible for a single instruction to extend over more than one editing record. The exception to this rule is the I instruction, used to insert text (see below).

CHARACTER STRINGS

If the user knows what character or group of characters he wishes to locate but is uncertain of their position within the file, the relevant record may be identified by means of a *character string.* A character string may be any character or group of characters enclosed by a pair of identical *string delimiters* from the following set:

: ; < = > ? ! " £ % & ' + /

Thus,

:STRING:

and

<STRING<

are valid character strings;

<STRING>

on the other hand is *not valid* because the delimiters are *not identical.*

Note: For the sake of convenience, character strings will be given the form

/STRING/

throughout this manual, unless otherwise specified.

THE POINTER

Understanding the *pointer* concept is essential in the use of the GEORGE editor. The point locates individual records or characters within records in the *oldfile.* Several editing instructions alter the position of the pointer. Many require that the pointer's location after the instruction has been executed be specified; for example, the T instruction transcribes a file up to (but not including) the point specified in the instruction. This point is called the *endpoint* and can be identified in several ways:

1 As an absolute decimal number; for example,

#2.4

The number to the left of the decimal point, preceded by a hash mark, respresents the position of the record relative to the beginning of the file. The second number, following the decimal point, represents the position of the character within the record. Both records and characters are numbered from zero. Therefore, the example above specifies as endpoint the fifth character of the third record of the file.

Note: . If the character field is omitted, the pointer will indicate the first character (character zero) of the record specified. Thus, if the endpoint is given as #2, the pointer will be set to the first character of the third record.

2 As a relative number;

7.5

indicates the sixth character of the seventh record from the record currently indicated by the pointer. For example, if the pointer is at #1.3, the above end-point will have an absolute position of #8.5 when the instruction is obeyed (note that the character field of the pointer's initial position is ignored).

3 As a character in a record containing a given character string:

C/FRED/.4

gives as endpoint the fifth character of the first record that the editor encounters which contains the string FRED.

4 As a character in a record beginning with a character string:

/SMITH/.3

specifies as endpoint the letter T, in the first record beginning SMITH.

5 As the (conceptual) character after the end of the last record in the file:

E

as endpoint will set the pointer to the beginning of a conceptual record after the end of the file; following a T instruction, for example, it will cause the file to be transcribed to the end of the last record.

E may also be used after the decimal point to indicate the character after the end of a specific record:

#2.E

sets the pointer to the character after the last character of the third record in the file.

In the above examples, the file or record is left open after the instruction is carried out; more text may be added to the record and additional records to the file. If a record is left open, the next instruction does not start a new record, even if the instruction occupies a new line in the editing file. A record is closed by moving the pointer to the first character of the next record, as in the instruction

T#3

which transcribes to the beginning of the fourth record, closing the third.

A file is closed by issuing E as an instruction; this transcribes to the end of the file, closes it, and ends the edit. (See the section which describes the basic editing instructions for more detail on the significance of

open and closed records.)

6 The record field may be null or omitted. This means that the endpoint is either located in the record containing the character currently indicated by the pointer, or, if the record field is specified as an absolute number (#0), is in the first record in the file.

The pointer is initially set to the first character of the first record (#0.0). Thereafter the user must take into account its changing position when he issues instructions, which will act with reference to the pointer's *current* position.

The remainder of this section deals with basic editing instructions which permit the user to transcribe (T) files or parts of files; to position the pointer without transcribing (P); to replace one character string with another (R); and to insert records or parts of records (I). The E and Q instructions, which end the edit, are also described, as is the F instruction which erases text already written to the new file.

Note: MOP users should see notes on editing in Chapter 5, page

Transcribing: T

This instruction copies text from the old file to the new file without alteration. The instruction is issued in the format:

T*endpoint*

where the *endpoint* is the character after the last character to be transcribed.

For purposes of illustration in this and succeeding passages assume that there is a file NAMES consisting of the following records:

Record number	*Record*
0	JOESMITH
1	JACKJONES
2	MOJAMES
3	HARRYEVANS
4	PETERBROWN

Unless otherwise specified, it will be assumed that the pointer is set to its initial position of #0.0 when instructions are issued.

The instruction

T#2

will transcribe NAMES up to, but not including, the character preceding the endpoint; that is, up to the beginning of the *third* record in the file. The new file will contain

JOESMITH

JACKJONES

If the user issued a second instruction

T2

this would transcribe from the pointer's current position (#2, the endpoint specified for the first instruction) up to but not including the first character of the second record from the current one, adding to the contents of the new file

MOJAMES

HARRYEVANS

Note: In the above example, the second T instruction begins a new record because the previous record has been closed (see section 5, page 22). If the first instruction had been

T1.E

the last record would have been left open and the effect of the second instruction would be

JOESMITH

JACKJONESMOJAMES

HARRYEVANS

Character strings can be used to specify an endpoint:

TC/JAMES/

will transcribe up to the first character of the first record containing the string /JAMES/, giving

JOESMITH

JACKJONES

while

T/JACK/

would transcribe up to the first character of the first record beginning with JACK giving

JOESMITH

Positioning the pointer: P

This instruction moves the pointer either forwards or backwards to a specified endpoint. (Moving the pointer backwards does not delete the intervening records in the new file, as the pointer refers only to the old file.) This facility could be used to alter the order of records within a file; for example,

T#1

P#4

T1

P#2

T2

P#1

T1

P#5

.

.

will transpose the fifth and second records of the old file.

DELETING

The P instruction may be used in combination with the T instruction to delete parts of the old file as it appears in the new file. Still referring to the file

JOESMITH

JACKJONES

MOJAMES

HARRYEVANS

PETERBROWN

The sequence

T#2

P#3

TE

(assuming that the pointer is initially at #0.0), will transcribe to the end of the second record, skip to the end of the third, and transcribe to the end of the file, giving

JOESMITH

JACKJONES

HARRYEVANS

PETERBROWN

Inserting: I

CHARACTERS

To insert a character or string of characters, an I instruction is given, followed by the text to be inserted. The text must be enclosed by string delimiters selected from the set listed on page 21. The delimiters should of course not be characters which occur in the body of the text. The text will be inserted before the pointer's current position.

In file NAMES the instructions

T#2

I/RUTH/

TE

will give

JOESMITH

JACKJONES

RUTHMOJAMES

HARRYEVANS

PETERBROWN

RECORDS

To insert a complete record or records, the pointer must be moved to the beginning of the record that is to follow the new record. The I instruction is the only instruction which can occupy more than one record in the editing file; if it were required to add three records to NAMES,

BILLCARTER

JANEBAXTER

MALCARDEN

after the fourth record, the I instruction could be used as follows:

T#3

I/BILLCARTER

JANEBAXTER

MALCARDEN

/,E

The format of the editing file records indicates the beginning and end of an inserted record; thus, a new line within an Insert instruction means a new line (that is, a new record) in the new file. The opening delimiter marks the position, with reference to existing records, of the beginning of the insertion, and the closing one locates the end of the insertion. Thus

T#3

I/BILLCARTER

JANEBAXTER

MALCARDEN/

E

gives

JOESMITH

JACKJONES

MOJAMES

BILLCARTER

JANEBAXTER

MALCARDENHARRYEVANS

PETERBROWN

while the instructions

T#3

I/

BILLCARTER

JANEBAXTER

MALCARDEN

/,E

will leave a null record before inserting the three new records.

Replacing data: R

The Replace instruction, which has the format

R/*oldstring*/*newstring*/

transcribes the current record up to the first character of *oldstring,* inserts *newstring* in the new file, skips *oldstring* and sets the pointer to the next character after *oldstring.* The two strings need not be of the same length.

For example, the instructions

T#S

R/EVANS/EBENEZER/

TE

will give

JOESMITH

JACKJONES

MOJAMES

HARRYEBENEZER

PETERBROWN

in the new file.

Note: When replacing a string by a longer or shorter string, users must take into account that the new record will be correspondingly longer or shorter than the old one.

The Forget (F) instruction

The Forget instruction erases anything written to the new file by the previous editing record, and sets the pointer in the old file to its position before this record was obeyed.

Note: The F instruction cannot be used twice without an intervening instruction.

Ending the edit: The E and Q instructions

The edit can be ended by either of two instructions: the End (E) instruction already described (see page 22) and the Quit (Q) instruction.

The End instruction transcribes to the end of the old file and ends the edit.

The Quit instruction closes the new file and abandons the edit.

ENTRANTS OUTSIDE THE FILESTORE

A basic feature of the filestore is that it is GEORGE, not the user, that controls the allocation of storage space to files. In certain circumstances this may not be convenient for the user. He may wish to ensure that a particular batch of information always stays on a particular magnetic tape or disc cartridge, so that it can easily be transferred to another installation. He may wish to control the shape of a direct access file to minimize head movement. To enable the user to control a part of backing store in this way, GEORGE allows conventional magnetic tapes and disc files to be included in the system. If a user wishes to hold information of more than 245K words in a single entrant, he must either use an entrant outside the filestore instead of a file, or make use of the *multifile* facility (see page 14).

There are two kinds of non-filestore entrant, *secure* and *insecure entrants.* They differ in the amount of control that GEORGE exercises over them. In the current mark secure entrants consist only of magnetic tapes. Insecure entrants can be either magnetic tapes or exofiles.

Secure entrants

GEORGE can keep, in the filestore, a record of some or all of the magnetic tapes used at an installation and can thereby provide a means of managing and controlling their use. The various files that contain these records and the routines for handling them are known collectively as the *Librarian.*

The Librarian maintains a record, in the file :SYSTEM.SERIAL, of all tapes known to the system. This record is consulted whenever a tape is requested, to ascertain whether the entrant is subject to Librarian security. Such entrants are of three classes:

1 Tapes owned by specific users

2 Pool tapes available for allocation to owners

3 Work tapes for temporary usage

Tapes are converted from pool tapes to owned tapes by means of the GET command and are returned to the pool by means of the RETURN command.

The user may also acquire pool tapes by means of a GETONLINE command, or a program may issue an unanticipated open mode #400 PERI, both of which bring the tape permanently under the user's ownership and also connect the tape on-line to the user's program.

The same security arrangements apply to owned secure tapes as to files in the filestore (see *Privacy,* page 19
When a tape is transferred to a user's ownership by one of the commands just described, a directory entry is set up in the owner's directory and the user trap system comes into effect.

Work tapes are kept in a worktape store and are used solely for temporary allocation to users. In the current mark worktapes are acquired in one of two ways. If the user wishes to pass the tapes between programs within a job, he issues a GET (or GETONLINE) command specifying a reference name in the form:

GET *worktape name*

The worktape name, which always begins with the character !, has no relation to the name on the tape's header label and serves only to identify the tape in order to connect it to subsequent programs within the job. Once the tape has been returned to the worktape store it can no longer be accessed by that name.

If it is not required to pass a tape between programs the user can obtain a worktape by an unanticipated #600 mode PERI or the equivalent ONLINE command. A tape obtained in this way can be given a worktape name by using the RENAME command.

The user may assign the same worktape name to several subsequent work tapes. These files are said to be *stacked;* when the name assigned to them is used, it will always access the most recently named worktape. If this tape is returned to the worktape store, the worktape named immediately before it will be accessed, and so on.

Insecure entrants

It is not always desirable for entrants to be objects of Librarian scrutiny since it may be necessary to move them frequently between installations. It is also necessary that facilities be available for accessing tapes which do not possess normal 1900 header labels. Facilities for these requirements are available in GEORGE with security arrangements reduced to the necessary minimum of ensuring that a secure tape is not accidentally used in this way. Magnetic tapes may under some circumstances be converted into secure entrants (see *Conversion of Entrant Categories*). In the current mark all direct access files outside the filestore are insecure entrants and are termed *exofiles.*

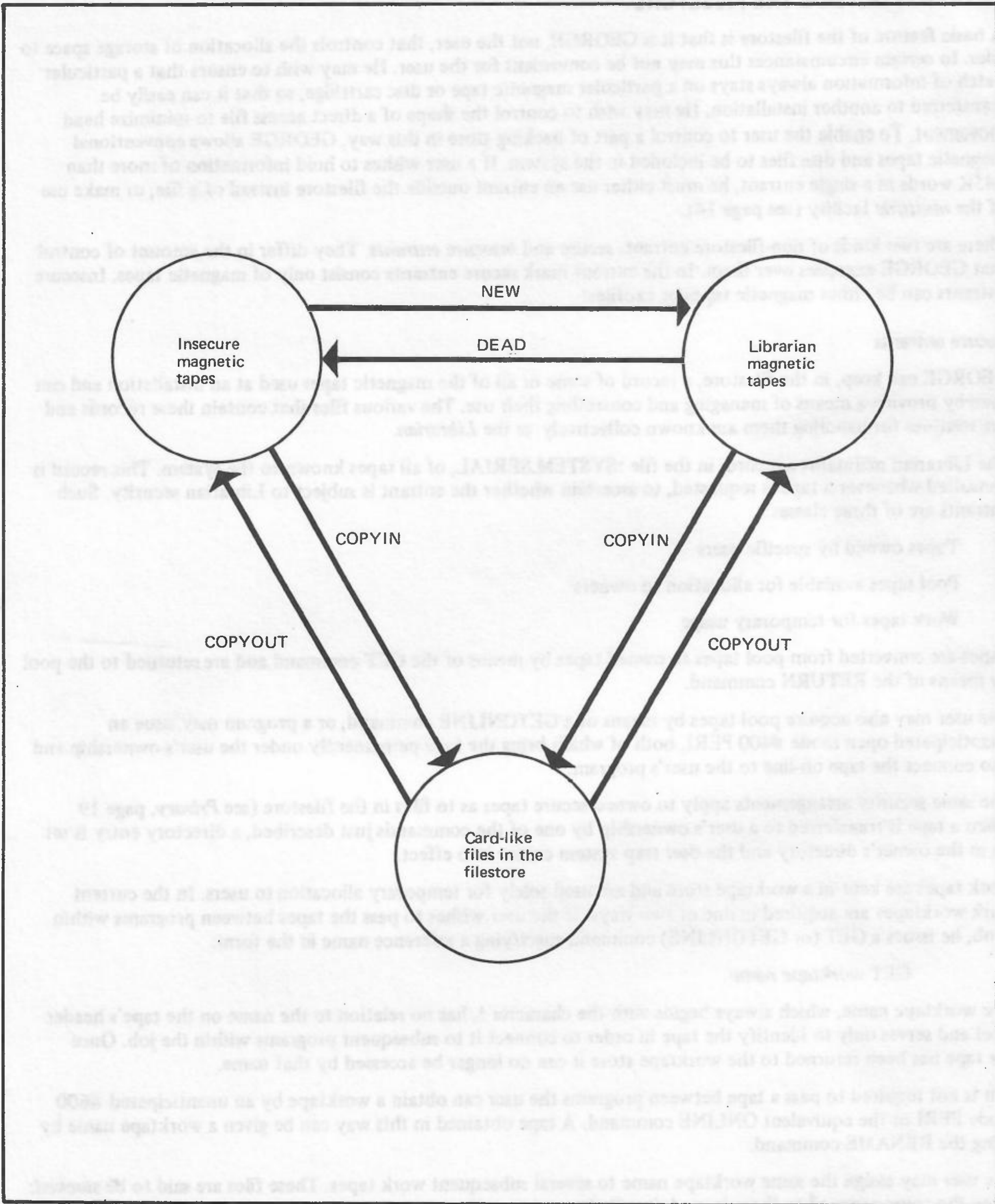


Figure 4 Entrant category conversion

An exofile is a conventional direct access file. These files, apart from scratch files, must be allocated by the same means as in a normal environment, since GEORGE commands are not available for this purpose (see the manual *Direct Access Utilities* (Edition 1, TP4190)). Once allocated, they can be connected to user programs by ONLINE commands or open mode PERI instructions in the same way as insecure magnetic tapes, except that the retention period for writing is checked.

Conversion of entrant categories

Entrants can be converted from insecure to secure entrants, since the data they hold need not differ in internal format. Both types of entrant can also be copied into the filestore.

The FILEIN and COPYIN commands copy magnetic tapes into card-like files in the filestore; the COPYOUT command copies basic peripheral files from the filestore to magnetic tape. These commands enter subject programs (system object programs) to copy data across, so that two versions of the data result.

Exofiles can be copied into the filestore using standard disc utilities (see the manual *Direct Access* (Edition 1, TP4107)) with appropriate job descriptions.

The NEW and DEAD commands convert entrants between the insecure category and the librarian category. In this case no copying is involved.

Figure 4 opposite illustrates the interaction of the conversion and copying facilities in diagram form.

Chapter 4 Input/output facilities

This chapter describes the various methods of handling input and output to user programs running under GEORGE. The GEORGE filestore, described in the previous chapter, enables the user to input his data from a basic peripheral to a file in the filestore before his program is run, and connect that file to the program at run time. Similarly, the user may output data from an object program to the filestore and have it listed on a basic output peripheral at any convenient time. In this way programs can be freed from dependence on basic peripherals, with a consequent increase in the throughput of the system.

There may, however, be cases where the user wishes to control basic peripheral input and output himself, as in an Executive environment, and GEORGE therefore allows basic peripherals to be connected directly to an object program. The user issues requests from his job description for the types of peripheral he requires, and can also specify particular properties that the output peripherals must have (see *The property system,* page 34).

INPUT TO THE FILESTORE

The INPUT command

In order to create a permanent file in the filestore, the user must input his data from a card reader, paper tape reader or a MOP terminal and head it with an INPUT command. The data following the command is read into a serial file in the filestore until a terminator is reached. The terminator is usually four asterisks, but if there are four consecutive asterisks included in the data, GEORGE will assume that they mark the end of the input data, so there is an optional parameter to the INPUT command allowing the user to specify another set of four characters as a terminator.

Example

```
INPUT :JONES,CARDDATA,T????

      lines of input data

????
```

This command will create a serial file named CARDDATA in the filestore and read into it the data following the INPUT command, up to but not including the four question marks.

PAPER TAPE INPUT

Where data is input from a paper tape reader, GEORGE has to know in what mode to convert the paper tape characters to internal six-bit characters. Mode parameters can be specified with the INPUT command and these correspond to the PERI paper tape modes normally available in the 1900 Series. The parameters ALLCHAR and NORMAL, for example, correspond to modes #26 and #06 respectively. (For a complete list of mode parameters, see *Operating Systems GEORGE 3 and 4.*) If no mode parameters are specified, a default value of NORMAL (corresponding to mode #06) is assumed by GEORGE.

Example

```
INPUT :JONES,TAPEDATA,ALLCHAR

      lines of input data

****
```

This command will create a serial file named TAPEDATA in mode #26 and read into it the data following the command up to and including the four asterisks.

Embedded INPUT commands

As an alternative to inputting data separately before a program is run, the user can include the data in his job description, preceded by an INPUT command. In this case the INPUT command and the data following it are read

into the job description file and the command is obeyed only when the job is run. (For details of job descriptions see Chapter 2). INPUT commands included in the job description are known as *embedded* INPUT commands. A disadvantage of this method is that the data has to be transferred twice, first into the job description file and then into its own file.

Embedded data

It is possible to include one or more streams of input data in a job description without any introductory INPUT command. In this case the data is read by the program direct from the job description file. Normally only one stream of data is input in this manner, as it is difficult to organize several interleaved streams.

INPUT AND OUTPUT FOR OBJECT PROGRAMS

The most common type of job run under GEORGE is one in which a program's input data is read into the filestore by an INPUT command before the job is run, and the output data is written to the filestore either to be output later on a basic peripheral or to be input to a subsequent program. In this case the peripheral transfers are said to be off-line. It is, however, also possible to have basic peripherals connected directly to a program so that data transfers can be carried out on-line.

Off-line peripherals: ASSIGN and LISTFILE

The provision of the filestore and off-lining facilities is one of the major advantages of the GEORGE operating system, and object programs should therefore make use of off-line input and output wherever possible. If peripheral transfers are all carried out off-line, the program is freed from restrictions imposed by the speed of the peripherals and the need for operator intervention to load and unload documents.

The ASSIGN command enables the user to connect the input and output channels of his program to filestore files, and he can obtain a listing of any appropriate basic peripheral file by issuing a LISTFILE command.

THE ASSIGN COMMAND

By means of the ASSIGN command the user can open a filestore file and associate it with a peripheral channel of an object program so that all PERI instructions for the peripheral are treated as PERIs for the filestore file. ASSIGN commands are included in the job description, preceding the ENTER command, and they connect the input and output channels of the program to the actual source and destination of the data. For example, suppose the user wishes to run a program held in a file called EXTRACT, which reads a pack of cards and prints certain items on the line printer. The cards have already been input off-line to the filestore and are held in a file called CARDDATA. The following instructions will cause the program to read the file CARDDATA, and write the results to a file called RESULTS.

```
JOB NO1JOB,:JONES
LOAD EXTRACT
ASSIGN *CR0,CARDDATA
ASSIGN *LP0,RESULTS
ENTER
ENDJOB
****
```

The input and output instructions in the program do not have to be altered in any way to take advantage of off-lining facilities. After an ASSIGN command has been issued, GEORGE will take care of any necessary changes in the format of data being transferred to and from the filestore.

THE LISTFILE COMMAND

When a program's output has been ASSIGNed to a file in the filestore, the user will frequently require a listing of that file and this can be produced off-line by means of the LISTFILE command. A LISTFILE command can be used to output any appropriate basic peripheral file in the filestore to any basic output peripheral or to the monitoring file (see page 10). The command must be issued from within a job description and the file it specifies will be listed as soon as an appropriate peripheral is available after the file is closed. If a program's output data file is closed before the end of the run, it is therefore possible for the file to be listed while the program is still running.

For example, if the user who submitted the program EXTRACT in the previous example wishes to have a listing of his output file on a line printer, he should amend his job description to read as follows:

```
JOB NO1JOB,:JONES
LOAD EXTRACT
ASSIGN *CR0,CARDDATA
ASSIGN *LP0,RESULTS
LISTFILE RESULTS, *LP
ENTER
ENDJOB
****
```

Note that the LISTFILE command is placed before ENTER. Since the ASSIGN command opens the file RESULTS, there is no danger of the file being listed before the program is entered. The advantage of placing LISTFILE before ENTER is that the output file may then be listed as soon as it has been closed (provided that a suitable peripheral is available), whereas if the order of these two commands were reversed, the LISTFILE could be implemented only after the program had been deleted.

If it is not necessary to list all the contents of the file, parameters may be included in the LISTFILE command to specify which parts of the file are to be output.

ERASING FILES

If a file will not be required after it has been listed, it can be erased by adding an ERASE command after the LISTFILE in the job description. Similarly, input files that will not be required after a program has ended can be erased by placing an ERASE command after ASSIGN in the job description.

For example, if the program EXTRACT is to be run once only, and the data will not be required after the results have been listed, the following job description could be used:

```
JOB NO1JOB,:JONES
LOAD EXTRACT
ASSIGN *CR0,CARDDATA
ASSIGN *LP0,RESULTS
LISTFILE RESULTS, *LP
ERASE CARDDATA
ERASE RESULTS
ENTER
ENDJOB
****
```

WORKFILES

An alternative method of storing data for once-only jobs is the use of *workfiles.* Workfiles are temporary files held in a *workfile stack* which belongs to the current job and exists only for the duration of the job. To set up a workfile, the user issues a CREATE command of the form:

CREATE !

where ! is always used as the workfile name.

A subsequent CREATE ! command will push down the workfile stack so that to access the first workfile the user will have to use the workfile name !1, since ! always refers to the file at the top of the stack.

For once-only jobs, workfiles can be used with embedded INPUT commands to eliminate the need to set up ordinary files and then erase them.

Example

```
JOB NO1JOB,:JONES
CREATE !
INPUT !,T????
lines of input data
????
LOAD PRINT
CREATE !       }
ASSIGN *CR0,!1 }  Pushes down the workfile stack so that the input file is now second
ASSIGN *LP0,!  }  in the stack and must be referred to as !1
LISTFILE !,*LP }
ENTER
ENDJOB
****
```

Note that the input data cannot be terminated with four asterisks as this would be taken as the end of the job description, so the optional parameter of the INPUT command is used to indicate that the input data will be terminated by four question marks.

On-line peripherals

Input and output operations can be handled via on-line peripherals in much the same way as in an Executive-controlled system. The peripherals that can be connected on-line to a program are basic input or output peripherals, magnetic tapes and direct access devices, and in all cases the ONLINE command is issued in the job description to connect the peripheral to the program.

BASIC INPUT PERIPHERALS

If data is to be read by a basic input peripheral on-line to a program, the user must preface his data with a DOCUMENT command, which gives a document name to the batch of data that follows. From within the job description an ONLINE command is then issued in the form:

ONLINE *peripheral name, document name*

GEORGE searches all peripherals of the type named until it finds the one on which the specified document is loaded. (If the document is not already loaded, it is requested on a suitable peripheral.) This peripheral is connected to the program channel specified by *peripheral name* and data transfers can then be carried out in the normal way.

BASIC OUTPUT PERIPHERALS

The ONLINE command for basic output peripherals has the same format as the command for basic input peripherals. In this case GEORGE will select a free peripheral of the required type and connect it to the program channel specified by the peripheral name parameter of the command. Output data will automatically be headed by the document name specified in the ONLINE command.

MAGNETIC TAPES AND DIRECT ACCESS DEVICES

Magnetic tapes and exofiles (direct access files outside the filestore) can also be connected to a program by the ONLINE command. GEORGE will perform certain checks to ensure that the correct tape or exofile has been loaded and that the user is permitted to access it, and will supervise exofile transfers (most data transfers between program and peripheral are handled by Executive).

THE PROPERTY SYSTEM

When a command requests a peripheral of a certain type, GEORGE normally selects any peripheral of the type requested that is currently free. There may, however, be cases in which one particular peripheral should be used to implement the command. For example, one of the line printers in an installation may be loaded with special

stationery and certain files must be listed on this stationery. The property system enables the user to specify the properties which the type of peripheral he requires must have.

The PROPERTY and ATTRIBUTE commands

Properties are given names of up to twelve characters, and these *property names* are declared to the system by means of the PROPERTY command. Once a property name has been declared to the system, it can then be connected with any peripheral on the installation by an ATTRIBUTE command. A peripheral may have more than one property ATTRIBUTEd to it, and each property may be ATTRIBUTEd to any number of peripherals. An optional parameter in the LISTFILE and ONLINE commands is used to indicate that the device allocated must have a certain property or properties.

Peripherals can be freed of previously ATTRIBUTEd properties by means of the CANCEL ATTRIBUTE command.

Some uses of the property system

The property system can be used to solve a very wide range of problems. The following examples show just a few of the ways in which it can be used:

1. Peripherals in a particular location may be given a property name that indicates their geographical position. Thus a line printer in Manchester, for example, can have the property MANCHESTER ATTRIBUTEd to it and a job whose output is required in Manchester can then specify that property in its LISTFILE commands. This use of properties is further described under *Peripheral clusters,* page 36.

2. When a maintenance engineer wishes to run test programs on a particular peripheral, he can ATTRIBUTE a property such as MAINTENANCE to the peripheral and so ensure that it is not used by any other job before being tested.

3. On some types of magnetic tape deck the assumed packing density of the tape can be varied by a manual switch. To cater for jobs which need to use these decks, property names such as 800PBI can be declared to the system and a job can then specify the packing density required by including the appropriate property name in its ONLINE commands. If GEORGE cannot find a deck with the property requested, it will output a message to the operator's console. The operator will then select a free deck, switch it to the required density and ATTRIBUTE to it the property specified in the job's ONLINE command.

4. If an installation has both fast and slow line printers and it seems that throughput could be improved by ensuring that large files are listed on the fast printer, FAST can be declared as a property name and ATTRIBUTEd to the fast printer. Jobs that output lengthy listings can then request a printer with the property FAST.

MULTIPLEXERS

A multiplexer is a device for co-ordinating simultaneous input and output between several communication channels and a single standard interface. Multiplexers may be connected on-line to user programs by means of the ONLINE command, in which case any kind of terminal device may be connected to the lines of the multiplexer, or they may be controlled by GEORGE for system use of MOP terminals and 7020 data terminals.

It is also possible to define a *conceptual multiplexer* which consists of a group of lines from any multiplexers or uniplexers on the installation that is to be treated by GEORGE in the same way as a hardware multiplexer. Several lines of a single multiplexer can therefore be defined as a conceptual multiplexer and connected on-line to a user program, whilst other lines of the same hardware multiplexer are being used by GEORGE for MOP or 7020 devices.

7020 REMOTE TERMINALS

The 7020 terminal is a communications device that allows on-line transfers of data between the central processor and a number of remote peripherals. Data is transmitted over a telephone line between a telephone terminal in the central computer room and the 7020 terminal. Basic input and output peripherals can be attached to the 7020. Each 7020 must have connected to it a 7023 teletypewriter which is used only as an operator's console. In this way a complete mini-installation for the input and output of batch data can be formed.

Facilities provided on 7020 terminals

User files and job descriptions can be read into the filestore from the 7020 by using the INPUT command in the normal way. Background jobs can be initiated from the 7020 by the use of JOB and RUNJOB commands.

The peripherals attached to a 7020 terminal cannot be individually on-lined to a user program, but the multiplexer or uniplexer to which the 7020 is attached can be used as an on-line unit; also the line to which the 7020 is attached can form a conceptual multiplexer or part of a conceptual multiplexer.

PERIPHERAL CLUSTERS

Groups of peripherals at remote installations are managed by the *peripheral cluster* facility, which is an extension of the property system. A *console property* is declared by a PROPERTY command including a parameter identifying a 7023 teletypewriter. This property is ATTRIBUTEd to each of the peripherals at the remote installation served by the 7023 console. A group of peripherals having a console property in common forms a *cluster,* which is given the name of the console property. When an ONLINE or LISTFILE command is issued, a console property name may be given as a parameter, in which case output from the command will be directed to the appropriate device within the cluster named by the console property. It is also possible, by means of an ASSOCIATE command, to specify that output from any LISTFILE command from a particular peripheral cluster be automatically sent to a given cluster. Usually this will be the cluster from which the command originates but if, for example, the user required an output device which his cluster does not contain, he could have output for that device routed to another cluster.

More information about the peripheral cluster system is to be found in Chapter 10 of the manual *GEORGE 3 Operation Management.*

THE OPERATOR'S FUNCTION

Servicing peripherals is one of the operator's main tasks in a GEORGE environment. GEORGE will output requests to the operator's console asking him to carry out any of the following actions:

1 Load input documents on to basic peripherals.

2 Load and unload magnetic tapes and exofiles.

3 Assist repeats on card and paper tape readers.

4 ATTRIBUTE properties to peripherals or free peripherals of properties previously ATTRIBUTEd.

In any of these cases there are two, or sometimes three, courses of action the operator can take:

1 He can implement the request.

2 He can issue a CANTDO command; this will cause GEORGE to take the default action appropriate to the activity that generated the request.

3 In the case of basic peripheral activities only, he can issue a TERMINATE command; this will terminate the activity that generated the request.

There is also a command WHATPER which the operator can use to find out the current state of a specified peripheral or all peripherals of a specified type.

MOP TERMINALS AS INPUT/OUTPUT DEVICES

A MOP terminal can be used to input data off-line to the filestore by issuing an INPUT command from the terminal and then typing in the required data, followed by a terminator. INPUT commands embedded in a job description (described on page 31) are more efficient when issued from a MOP console than from a basic peripheral. Since each command typed on the console is obeyed immediately, rather than being first stored in a job description file as in a background job, data following an embedded INPUT command is transferred directly to a filestore file, bypassing the intermediate stage of being stored in the job description file.

It is, however, more common for MOP terminals to be used as on-line input/output devices. If the user issues an ONLINE command without a document name parameter from his console, PERI instructions for the peripheral specified in the command will be implemented either by reading data typed in by the user or by sending output to the console as appropriate.

The MOP facilities available under GEORGE will be fully described in the next chapter.

Chapter 5 MOP

INTRODUCTION

For a powerful computer with a capacity for a large work load, it is essential to reduce to a minimum the amount of human intervention at run-time, to enable this capacity to be fully utilised. This can be done by planning each stage of a process beforehand and presenting the operating system with a complete job description to control the run.

Some types of work are not suited to this kind of operation, but require decisions at various stages of the run, because each decision depends on how the process has behaved so far. For this type of job it is best if the step-by-step decisions can be made by the originator of the job, rather than by the computer operator. If the various courses of action can be described to the operator, they can usually be described equally well to the operating system. What is required is the facility to 'converse' with the computer via an on-line terminal, using the same command language as is provided for batch jobs under the control of GEORGE.

The GEORGE MOP system provides this facility. It allows a number of users to have simultaneous access to a computer, while batch jobs are being processed as background. MOP shares computing time between the various on-line users so that each user has the illusion that the entire machine is at his disposal. This is possible because, for each on-line job, processor utilisation is low, the most time-consuming elements being the human decision, the input of commands to the operating system telling it what to do next and the output of responses from the system. Since each on-line job makes only a small demand on the computer, the operating system can run a larger number of 'conversations' simultaneously.

ENVIRONMENT

The current version of MOP is designed for use with 7071 console typewriters. Later versions will cater for other communications devices. Users are also able, by means of the peripheral cluster system, to run batch jobs via a remote configuration of peripherals (card or paper tape readers, line printer, or MOP terminal linked to a central processor by telephone lines (see page 35)).

The minimum configuration required for the current version of MOP is given on page iii. A core size of at least 96K words is recommended, although MOP can theoretically be used with less.

SYSTEM CONTROL OF ON-LINE USE

Each MOP user has access to the filestore and is subject to the filestore controls described under *Accounting*, page 19. In addition, before a user is allowed to initiate an on-line job, GEORGE may carry out a password check to ensure that the user is authorised to use the computer.

A user starting an on-line session with GEORGE first issues a LOGIN command, giving his user name; for example:

LOGIN *job name*, :FRED

GEORGE first checks that the computer is capable of bearing the load. There is an installation parameter (JOBLIMIT) that sets a limit on the number of jobs of all kinds that may be introduced to the system. The value of this parameter is originally calculated by the Early Morning Start routine of GEORGE but may be reset by the command INSTPARA. (Further details may be found in the *GEORGE 3 Operation Management* manual.) If the initiation of this job would cause the limit set by INSTPARA to be exceeded, the command is rejected.

GEORGE then searches its Dictionary (see page 15) and, provided that it finds an entry for this user, it asks him to give his password. This is compared with the one held in the Dictionary against the user's name. If the correct password has been given, the user is allowed to run a job.

A simple check is carried out on the state of the user's money budget (see Chapter 7) when he tries to start a job. If he has used more money than is allocated to him, an error is signalled and the LOGIN command is abandoned.

Once logged in, the user can issue standard GEORGE commands from the MOP terminal. Each command is obeyed as soon as it is given, and the user is then invited to issue his next command.

THE BREAK-IN FACILITY

An important feature of MOP operation is the *break-in* facility. By means of a special signal the MOP user can freeze the operation currently in progress in his job and return the system to a state such that it is ready to receive a command.

The user can break in before, during or after the implementation of a command or during the running of a program. Later he can continue his job from the point at which the break-in occurred, by issuing a CONTINUE command.

While the user is broken-in, he can issue any command that does not create a core image or enter a program. The user can, for example, break-in during a program run, and examine and alter the core image. Having made changes to the core image, he can issue a CONTINUE command, and the program run will continue from the next instruction as indicated by the contents of word 8.

Break-in levels and command processor levels

As was explained in Chapter 2 (*Command processor levels*, page 8), the command processor level of a job normally changes when the source of commands changes. This is equally true of MOP jobs. When the user logs in, his job is initially at command processor level zero. The job remains at level zero until the source of commands changes from the MOP terminal to a macro, a program or a built-in command; the level of the job then changes to one. Further command processor levels may be created in the same way.

If the user breaks-in while his job is at a command processor level greater than zero, all existing levels are preserved and a new command processor level is created. This level is both a command processor level and a *break-in level.* Because it is a break-in level, it is logically distinct from all the command processor levels created before the break-in. What this means in practical terms is that the only way the user can return from the break-in level to the command processor level above it is by terminating the break-in by means of a CONTINUE command.

During a break-in, command processor levels can be created in the normal way, for example by macros. These levels will be break-in level one, break-in level two, and so on. The user can terminate the break-in by issuing a CONTINUE command at any of these levels, When the CONTINUE command is given, all the existing command processor levels up to the top level are destroyed, not merely the levels created during the break-in. In addition, the current core image is deleted if there is one.

If a break-in signal is given during a break-in, all the existing break-in levels are destroyed, but the job does not continue from the point at which the original break-in occurred (unless of course a CONTINUE command is given). Instead, the job remains broken in, and a new break-in level zero is created.

MONITORING WITH MOP

As was explained under *monitoring files,* page 10, a monitoring file is created for each job that is run under GEORGE. This file contains categories of information specified in a TRACE command. In the case of a MOP job, the user can arrange for some or all of the information that is sent to the monitoring file to be output at the MOP terminal during the course of the job. This is achieved by means of the REPORT command.

Using this command, the user can specify that certain categories of monitoring data, such as object program output, command errors or log analysis information, are to be output at the MOP terminal. During the job, the user can adjust the amount of monitoring data output, by issuing further REPORT commands. In addition, at the end of the job the user can get a complete or partial listing of the monitoring file on a line printer, by means of optional parameters in the terminating command (LOGOUT).

TYPICAL MOP OPERATIONS

One standard application of MOP, which illustrates many of the facilities it offers the user, is program development: the preparation, compilation and testing of programs. A programmer wishing to test a program using MOP facilities must first log in to the system in the manner described above (page 37). Provided that he is acceptable to GEORGE, he can file his source program in the filestore by inputting it from the MOP terminal. If the program is long, he may prefer to input it to the filestore on cards or paper tape, in order to save himself the trouble of typing it line by line.

Normally every source program that is tested on a particular installation will have been written in the source language of one of the compilers provided with the operating system or added by the installation itself. Provided that this is true a system macro will have been filed in a system file, to provide the necessary interface between GEORGE and the required compiler. To compile his source program the programmer must simply type the correct system macro command, together with parameters giving the names of the appropriate input and output files and any special options needed. This command will call in the compiler from a system file and enter it. Compiler diagnostics can be output on the MOP terminal as they occur. If errors occur during the compilation, the user may either abandon this run of the compiler at once, or allow the compilation to finish and then call the Editor.

To abandon the compilation, he must break-in using the break-in signal and then issue a QUIT command to delete the existing command processor levels. He can then call the Editor to amend his source program.

To edit the contents of a file he must issue an EDIT command (see Chapter 3). This command calls in the built-in Editor which will edit the file line by line in accordance with special editing instructions. These instructions may either be issued from the MOP terminal or be stored in an amendments file and issued from that file.

The user may have the contents of the (edited) file output on his MOP log by requesting a listing at the time of issuing his first editing instruction, in this format:

L, *instruction*

After each instruction, the text in the newfile is also typed out on the log.

Note: Incomplete records are not output; an instruction to transcribe two records and half of a third will cause only the first two to be output on the log. The third will be output when it has been closed by a subsequent instruction (see Chapter 3).

The programmer will normally have specified that a core image of the object program is to be produced. He may, however, have specified that a file containing semicompiled segments is to be produced so these segments may now be consolidated with other semicompiled segments and loaded. Before entering the object program, the user must inform GEORGE how the program's input and output instructions are to be handled. As with background jobs, the program may be linked to permanent input/output files, temporary input/output files or on-line peripherals. Alternatively, input and/or output instructions may be associated with the MOP terminal (by ONLINE commands). In this case, when the program requires input, the programmer will be invited to input a line from the MOP terminal, and when program results are ready for output they will be sent to the MOP terminal instead of being output on a basic peripheral or written to the filestore.

If the program comes to a natural conclusion, GEORGE will indicate that it is ready for another command. If the program does not end naturally, it will have run until either the user has quitted the program, because, for example, it seems to have got into an endless loop, or a program event has occurred. Program events include program failures (due to illegal instructions or reservation violations, for example), program halts or deletions, or a failure due to a program exceeding a time limit. (By means of a TIME command, the programmer can set a time limit on a program run to ensure that he does not waste processor time on a program that has got into an endless loop.)

In all of these cases the program becomes *dormant* and is preserved in its unfinished state until the programmer types another command. If he wishes, the programmer may file the unfinished program as a *saved-program file* using a SAVE command. He will then be able to restore the program later and resume it from the point reached.

The programmer may at this stage wish to perform a post-mortem on his program or rerun selected portions of it in a regulated manner, for example under the control of a MONITOR command. This command will allow the programmer to stop the program at a pre-arranged points to examine the values of various program variables and registers, and then to alter individual instructions and restart the program at an arbitrary point. In this way the programmer can test the program until either it runs correctly or an error is found that necessitates recompilation.

FURTHER MOP FACILITIES

Background jobs

Normally, the MOP user wishes to control his job from the MOP terminal, issuing commands one at a time and waiting for each one to be obeyed before issuing the next. However, there are likely to be times during an on-line session when little or no communication between man and machine is required. For example a programmer who has finished testing a long program on-line and now wishes to run it using input data held in the filestore, does not usually expect to have to intervene during the run. GEORGE allows the on-line user to initiate a background job from his MOP terminal. The user can issue a command involving a long process, or a series of commands filed as a user defined macro, and can then allow the job to proceed autonomously. While the job is being run as background, the user can perform other MOP operations. Later he can, if he wishes, resume his interaction with the first job or link some other background job to the MOP terminal.

Conversing with an object program: The ONLINE command

Instead of conversing with GEORGE, the MOP user may converse directly with an object program under GEORGE's control by connecting input and output channels of the program to the MOP terminal. The user issues an ONLINE command and identifies the peripheral device at which the relevant data transfers are requested. Transfer requests involving this peripheral will subsequently be directed to the MOP terminal, and data can be typed in from or output to the terminal even though the program's original transfer requests refer to basic peripherals. For example, the command

ONLINE *CR0

means that each request for input from card reader zero will result in an invitation to the user to type in data from the MOP terminal.

ONLINE *CP0

causes output which the object program would normally direct to card punch zero to be printed out on the MOP log.

Subsystems under MOP

As has been explained, it is possible to incorporate almost any item of software in the operating system by means of a system macro command. In particular it is possible to make available to the MOP user any special purpose subsystem designed for on-line use: for example, one requiring remote and immediate access to a filing system.

EXAMPLE OF A MOP JOB

The following is an example of a MOP job. It demonstrates the sort of situation in which the usefulness of an on-line terminal is most obvious: the user attempts to compile a program and the compilation fails. He is then able to request that the program be listed on his MOP log; he locates a syntax error, corrects it using the MOP editor facility, compiles the program successfully and runs it.

←LOGIN FRIARMJ, :AUTH	
TYPE PASSWORD←XXXX	User logs in
STARTED :AUTH,FRIARMJ,2JAN71 15.06.27	
←QFORTRAN BUG	Requests compilation of FORTRAN program in file BUG
CORE GIVEN 18176	
0.11 :HALTED : LD	
HALTED ZZ	Halted ZZ indicates failure to compile
.	
.	
.	
15.09.36← LISTFILE BUG,NUMBER	Lists the contents of BUG numbering records
1 LIST	
PROGRAM	
.	
.	
11 RBUGAD(1,20)BLANK,COMMA,FSTOP,Q	Syntax error in record eleven: 'RBUGAD' for 'READ'
.	
24 STOP 444	
25 END	
26 FINISH	

****	End of listing
15.11.28←EDIT BUG,OKPROG	Requests editor to edit BUG into new file OKPROG
EDITOR IS READY	
0.0←T11	
11.0← R/BUG/E/	Alter RBUGAD to read READ in OKPROG
11.10← E	Transcribes remainder of BUG into OKPROG and ends the edit
15.13.28← QFORTRAN OKPROG	Second request to compile
15.14.00 0.16 CORE GIVEN 18176	
0.18 :HALTED : LD	
0.20 :DELETED : FI #XPCK	
.	
.	
0.28 :HALTED : LD	Successful compilation; program loaded
15.16.00← ONLINE *CR0	
15.16.09← ONLINE *CR1	Input/output channels connected to MOP terminal
15.16.18← ONLINE *LP0	
15.16.38← ENTER	
TYPE NECESSARY CONSTANTS	
← .	
.	Program running
.	
DISPLAY : 444	
0.30 :DELETED : 00	
15.19.18 0.30 DELETED	
15.19.21← LOGOUT	End of run; user logs out

Chapter 6 Scheduling

SCHEDULING

In a 1900 Series installation that is not controlled by an operating system, the task of scheduling is shared by the operator and Executive. To make use of the multiprogramming facilities of the larger 1900 Series computers, the operator loads a number of programs into the available core store and Executive shares out central processor time between these programs in accordance with their priorities. Executive always allows the program with the highest priority to use the central processor, provided that this program is free to run. When a program is suspended, because, for example, it is awaiting the completion of a peripheral transfer, the program with the next highest priority is entered. At the end of the transfer, Executive is called in to reactivate the top priority program. Whenever a program is completed, the operator is informed. He can then load a new program into the area of core-store that has become available.

This method of scheduling in a multiprogramming environment tends to be inefficient for a number of reasons. Firstly there are bound to be long delays when the operator loads each new program as space becomes available.

Secondly it is the operator's task to decide which program to load at any one time. To achieve an ideal job mix he must take into account not only the amount of available core but also such factors as the peripheral requirements of the various programs he has to run. Because of the complexity of the factors involved, the job mix is likely to fall far short of the ideal and throughput consequently suffers.

Thirdly the system of Executive priorities is inflexible. The operator has no means of ensuring that a low priority job uses the central processor at all, other than by adjusting program priorities. Where all the work involved is background work, this inflexibility is not very serious, though it does make it difficult for the operator to ensure that all deadlines are met. Where some or all of the work is on-line, the Executive-controlled system is inadequate. It is essential that a fast response be given to all on-line users, but a simple priority arrangement cannot guarantee this.

The two GEORGE routines that control scheduling are called the high level scheduler and the low level scheduler. At any given time there may be a number of jobs loaded and waiting to be run. The function of selecting which of the waiting jobs to run is performed not by the operator but by the high level scheduler. This routine makes its selection on the basis of the urgency, peripheral needs, and so on, of the jobs waiting in the filestore, and the scheduling requirements that have been presented to it by users. The jobs that are selected by the high level scheduler are passed on to the low level scheduler. This routine shares out central processor time in accordance with policies presented to it by the high level scheduler.

THE HIGH LEVEL SCHEDULER

Factors relevant to high level scheduling

The high level scheduler can be thought of as formulating the strategy of the scheduling process, by determining the policies that the low level scheduler carries out.

The purpose of the high level scheduler is to control the supply of jobs (in particular, background jobs) to the low level scheduler and to regulate the amount of machine time used on each one, in order to maintain efficient operation of the system. It has under its control all the jobs that have been submitted to the system. It also has information about the state of the system, such as the size of the core store and the availability of peripherals, and information about each job, such as its time of arrival and urgency (GEORGE priority). Different installations may require a variety of different constraints to be placed on the way jobs are scheduled. For this reason a large part of the high level scheduler consists of a subject program.

Factors that might affect overall scheduling at an installation include the size of core store available for running programs, the proportion of computing power allocated to background rather than MOP jobs, and the availability of peripherals.

In addition to these, there are such factors as the following ones which affect the scheduling of individual jobs: the urgency of a job and the time before which it must be completed, a job's estimated run time and program size, the

availability of required files and whether the job is limited by the central processor, the filestore or on-line peripherals.

The above is not an exhaustive list of the factors that may be relevant to an installation. They merely serve to show some of the things that could be taken into account in determining high level scheduler policy. In an installation where the majority of jobs are test runs of programs, preconceived estimates of job characteristics may prove to be pious hopes rather than a reliable guide for scheduling. In such an installation it is probably best to take into account only user requirements such as urgencies and deadlines. Other installations may wish to experiment with a variety of methods of dealing with job characteristics in order to aid scheduling.

What the high level scheduler does

Note: While a standard high level scheduler is issued with GEORGE 3, installations may write their own HLS to suit their particular requirements, but the following information is likely to apply in most cases.

The user can give the high level scheduler details about a particular job and its scheduling requirements by means of commands issued from a MOP terminal or included in a job description. These commands correspond roughly to those of the factors listed above which can be specified by the user in the scheduling of individual jobs.

Thus, there is a command called the URGENCY command, which assigns a GEORGE priority (as opposed to an Executive priority) to a job. The urgency of a job is expressed as a letter of the alphabet between A and Z, A being the highest urgency, Z the lowest. Another command that may be used to communicate scheduling requirements to the high level scheduler is MAXSIZE, which sets an upper limit to the size of core images within the job. There is also the JOBTIME command, which has the dual purpose of informing the high level scheduler of the amount of mill time used by a job and setting a limit on it. In its second function this command operates independently of the high level scheduler.

Using the data at its disposal the high level scheduler decides which jobs to run (in practice, which jobs to tell the low level scheduler about). Typically, most if not all the MOP jobs will be accepted, but not all the background jobs. To prevent the system being overloaded with MOP jobs, an upper limit is set on the number of MOP users who can run programs *interactively*. After the user has logged in, his job can be *tentatively started*, that is, it will be allowed to proceed until a command such as LOAD, CREATE, ASSIGN, SAVE, REALTIME, ENTER, ONLINE or RESUME is encountered. If the limit for interactive programs has been reached, the user will be informed that his job cannot be *fully started*. Jobs must be made fully started by the high level scheduler before they can run programs. If a particular job is of a very high priority the EXPRESS command can be used to inform the high level scheduler that this job should not be made to wait when it requires to be fully started, and that a high proportion of computer time should be devoted to it.

The next step for the high level scheduler is to examine the relative urgencies which users have set on the selected jobs and to establish a *Computing Power Index* (C.P.I.) for each one. The C.P.I. for a given job is a number representing the proportion of the computer's time to be given to that job. It is not an absolute, unchanging number, but is relative to the urgencies of the other jobs that are to be time-shared. As jobs start or finish, the C.P.I.'s for other jobs may need to be changed. To achieve this dynamic control the high level scheduler is called in at regular intervals and also whenever a new job is submitted to the system or commands that necessitate scheduling action are given by a user (for example, if a user issues a new URGENCY or MAXSIZE command, thus changing the job's scheduling requirements, the high level scheduler is called in before a program can be entered or resumed by this job).

To illustrate the meaning of the C.P.I., consider the case of three jobs to be run with urgencies F, K and Q respectively. The high level scheduler might assign C.P.I.s which informed the low level scheduler that the F job was to run for 48 minutes in every hour, the K job for 10 minutes and the Q job for 2 minutes. This does not mean that the F job could run for 48 minutes continuously, if it was long enough, because the C.P.I. is used to determine the proportion of computing time given to a job, not the length of time for which a job can run without interruption. This latter factor is controlled by the low level scheduler.

THE LOW LEVEL SCHEDULER

The low level scheduler is concerned with the tactics of scheduling. It arranges the sharing of time between programs in accordance with the C.P.I.s presented to it by the high level scheduler. It is automatically entered several times every second to decide which program to run next.

Under GEORGE more programs can be active than there is room for in core. Less active programs are kept by the low level scheduler on backing store. The low level scheduler assigns to each program a *time slot* and a *fair waiting time*. A time slot is the amount of time a program may run without interruption. Its length, for both background and MOP programs, is proportional to the size of the program. The actual value is determined by the value of the installation parameter SLOTTIME in combination with the core size.

The fair waiting time is the amount of time a program should ideally wait between time slots, and is determined by the program's C.P.I., the number of programs currently loaded and the actual time used in the current time slot.

Each time the low level scheduler is entered at the end of a program's time slot, the next program to be run will be the one for which the actual waiting time exceeds the fair waiting time by the greatest amount (or falls short of it by the least amount). If this program is on backing store in the filestore, the low level scheduler first loads it into core. This process may involve *swapping* some other program out of core and dumping it to backing store. When only background jobs are being run, the low level scheduler will not swap out a program at the end of its time slot in order to replace it by another program unless it is necessary to relocate programs in order to make the optimum use of core store. It is therefore unnecessary, and not recommended, that the value of SLOTTIME be set unduly high in this situation.

EXECUTIVE SCHEDULING

Executive carries out the scheduling process at the millisecond level, as in the normal Executive-controlled system. If a program is suspended during its time slot, the system is not held up awaiting the next entry of the low level scheduler. Executive automatically switches control to the program in core with the highest Executive priority. In future versions, to make this process as efficient as possible, the low level scheduler will dynamically adjust Executive priorities at frequent intervals, to ensure that the right program is entered by Executive.

Chapter 7 Budgeting and accounting

BUDGETARY CONTROL

Because of the hierarchical structure of the filestore, it is possible for the installation manager to keep strict control over the use that is made of the computer.

A user with a directory in the filestore and who holds the NEWUSER privilege can, by means of the MAKEDIR command, create users with directories immediately inferior to his proper directory. Each user represents an account, so that, in creating new users inferior to his directory, a user is effectively breaking down his account into separate categories. In this way a manager-user in charge of a number of different projects can arrange for each to be costed separately and can determine in what proportions the computing facilities at his disposal are shared between them. The user name of each new user may or may not denote an actual individual. It may serve to identify one of a number of projects under a single man's control or it may denote a group of people associated with a single project. In either case the account might be broken down further, into sub-projects, and so on to a depth of up to 64 levels below :MASTER.

Each user in the system is allocated *budgets* by his immediate superior and can himself allocate some or all of his budgets to users he has created, the amounts he gives to his inferiors being subtracted from his own budgets. There are three main budget categories, space, time and money. Budgets are allocations of space, time and money. Space budgets are allocations of space for entrants of different kinds and are measured in space units, for example magnetic tapes. Time budgets refer to central processor time for use in running jobs of different urgencies (A to Z). The urgencies for which time budgets are to be allocated and the units assigned to each are decided by the installation manager. A money budget is the quantity of money, real or notional, that is allocated to pay for a user's consumption of time and space-time units, according to the charging algorithms in use at the installation. In the current Mark there are four budget types:

SPACEMT	measured in units of one magnetic tape
TIME	measured in units of one second of mill time
MONEY	at the installation manager's discretion
REALTIME	measured in words of core for realtime programs (see below *Classification of budgets*.

CLASSIFICATION OF BUDGETS

Budget types are classified as *transient* or *stable*. A transient budget is one that is permanently diminished by being used, a stable budget is one that can be made re-available after use. Time and money budgets are transient, whereas space budgets, including the REALTIME budget which limits the amount of core available for realtime work, are stable.

Transient budgets

Transient budgets are allocated to users at intervals, for use during the coming *budget period*. A budget period is the time interval between entries of the accounting program; its length is determined by the installation manager.

In the case of transient budgets, a distinction must be drawn between a user's *ration*, the amount of time or money regularly allocated to him for each budget period, and his *allowance*, the amount that has been made available to him for the current budget period. When a user runs a job, he consumes some of his time and money allowances, but he does not alter his rations of time and money. If a user gives away part of his money ration, he does not affect his money allowance for the *current* period, though his money allowance for the *next* period will be less as a result.

A user's ration of a particular transient budget type may or may not be equal to his allowance. This will depend on whether the user's ration (or allowance) is altered in the middle of a budget period and also on whether the user's consumption during one period affects his allowance for the next period.

Stable budgets

In the case of stable budgets, the term 'allowance' has no meaning, an allocation of space being called a ration. Since stable budgets are not permanently diminished by being used, there is no need for a user's allocation of space to be replenished at periodic intervals. Once a user has been given a space ration it belongs to him permanently, until he decides to give it away or it is taken from him by his immediate superior.

The difference between the ways stable and transient budgets operate can be illustrated by an example. Suppose that a user has a stable budget ration of one hundred magnetic tapes, of which he is using ninety at the moment. Should he now require fifteen tapes, he can release five of the tapes he is using and thereby increase the number of tapes available to fifteen.

In the case of a transient budget like money, this kind of operation is not possible. If a user has an allowance of £100 and he has used £100, he cannot recover any of the money he has spent. He must either supplement his allowance by using part of another user's allowance (see *Removal of budgets*, below) or wait until his allowance is renewed at the beginning of the next budget period. The size of his money ration does not affect the amount of money he can use in the current budget period; this is used to determine the size of his new allowances at the beginning of the next period.

ALLOCATION OF BUDGETS

The standard method of allocating budgets is by means of the BUDGET command. This enables a user to give stable and transient budget rations to users that are immediately inferior to him. The amounts that he can allocate are limited by the sizes of his own budget rations, since it is these that he is in effect giving away. A user can also allocate rations to his immediate superior by means of this command, but he cannot give rations directly to any user more than one level above or below him in the hierarchy.

Since the BUDGET command acts only on rations, it cannot affect a user's allowance of time or money for the current period. A user whose money ration has been increased will begin to benefit only when transient allowances are reallocated at the beginning of the next period. To increase a user's current allowance of time or money, an ALLOWANCE command must be given. Unlike BUDGET, this command can be used to increase the allowance of any user in the system, but the amount given is subtracted from the giver's current allowance. The virtue of this facility is that if a user finds that he requires an abnormally large money allowance for a special project during the current period, a user any number of levels above (or below) him in the hierarchy can increase his allowance directly without affecting the size of his regular ration.

REMOVAL OF BUDGETS

A user's power to reduce other user's budgets is more limited. By means of the BUDGET and ALLOWANCE commands, he may take back rations and transient budget allowances only from users who are immediately inferior to him. This restriction is sensible enough. Clearly it would be impossible for a line manager to organize his budget efficiently, if his managing director was liable at any time to appropriate budgets belonging to the line manager's staff. Under the GEORGE budgeting system a managing director who wishes to cut down on his company's budgets would have to decrease the budgets of his line managers. They would then be responsible for the efficient reallocation of their reduced resources.

PRIVILEGES

To aid the installation manager in controlling his installation, GEORGE provides special kinds of budgets, the *privileges*. Once assigned a privilege by the manager, by means of a BUDGET command, a user can make use of the facilities specified for that privilege in the file :MASTER. DICTIONARY, while users not holding the privilege restricts the right to create new users by means of the MAKEDIR command, while the TRUSTED privilege is needed if the user is to perform extracodes requiring a trusted status for the object program using them.

NEWUSER and TRUSTED are examples of *built-in privileges*; of the twenty privileges available in :MASTER.DICTIONARY, a small number are of this type; the rest may be defined by the manager in accordance with the needs of the installation.

Users holding privileges may assign them to their immediate inferiors in the hierarchial structure, or take them away by a BUDGET command with a GIVE or TAKE parameter. If a user loses a privilege his inferiors also lose it.

At the end of a budgeting period, the installation manager runs an accounting subject program to compile period accounts for each user and to allocate fresh transient budgets. This aspect of GEORGE's budgeting and accounting facilities is more fully described in the Operational Management manual.

Appendix 1 Writing a job description

This Appendix describes the organization of a typical job description to compile and run a program. It should be read in conjunction with Chapters 2 and 4 of this manual.

THE COMPLETE JOB DESCRIPTION

It is assumed that the source program and the input data have already been input to the filestore as follows:

```
INPUT :JOHN,PROGA
lines of source program
.
.
.
.
****

INPUT :JOHN,PROGADATA
lines of data
.
.
.
****
```

The job description can then be submitted to GEORGE:

```
JOB MYJOB,:JOHN
WHENEVER COMMAND ERROR, GO TO 2L
LINGO PROGA
SAVE BINPROGA
ASSIGN *CR1,PROGADATA
ASSIGN *LP1,PROGAOUT
LISTFILE PROGAOUT,*LP
ERASE PROGADATA
ERASE PROGAOUT
ENTER
IF HALTED, RESUME 21
IF FAILED, PRINT (0,7)
IF DELETED "PROGA OK", RUNJOB NEXTJOB,JDFILE
2L ENDJOB
****
```

INTRODUCING THE JOB DESCRIPTION TO GEORGE

If the job is to be run more than once, the job description will be introduced to GEORGE by an INPUT command so that it can subsequently be initiated by a RUNJOB command. If, however, it is only necessary to run the job once, the job description will be introduced by a JOB command. In this appendix it is assumed that the job will only be run once, so the first command in the job description is:

JOB MYJOB,:JOHN

THE WHENEVER COMMAND

WHENEVER commands can be issued at any point in the job description. In the example given here, the WHENEVER command at the start of the job description ensures that if there is an error in any of the commands the job is immediately abandoned.

COMPILING THE SOURCE PROGRAM

The job's first task is to compile the source program, so the next command in the job description will be a compilation macro. In this example, the fictitious macro LINGO is used, and its format is assumed to be:

LINGO *source program file name*, *object program file name*

If the second parameter is omitted, the object program will be left in core at the end of the compilation. A listing will always be sent to the monitoring file system.

The command included in the job description will be:

LINGO PROGA

This will compile the source program in the file PROGA and leave the binary output in core ready to run. If the user wishes to keep a copy of the object program he then issues a SAVE command:

SAVE BINPROGA

A card-type file called BINPROGA is created in the filestore by this command, and the current core image copied to the file. If the user wishes to run the program again later, he can write a job description starting:

JOB NEWJOB,:JOHN

RESTORE BINPROGA

CONNECTING PERIPHERALS TO THE PROGRAM

Before the object program in core can be entered, the user must connect the program's peripheral channels to filestore files. This he does by means of ASSIGN commands:

ASSIGN *CR0,PROGADATA

ASSIGN *LP0,PROGAOUT

If he wants a listing of the output file, he can then issue a LISTFILE command:

LISTFILE PROGAOUT,*LP

ERASE commands can be placed after the LISTFILE if the input and output files will not be required again:

ERASE PROGADATA

ERASE PROGAOUT

The files will not be listed and erased until after they are closed (see page 33).

ENTERING THE PROGRAM

When all the peripherals have been ASSIGNed, the program can be entered. If the entry point is zero, the command is simply:

ENTER

PROGRAM EVENTS

After ENTER, the user places the conditional commands which inform GEORGE what action to take if a *program event* occurs. In this example of a simple job description, three types of program event are monitored.

1 HALTED EVENTS These are caused by the program halting

2 DELETED EVENTS These are caused by the program being deleted

3 FAILED EVENTS These are caused by program failures such as illegal instructions and peripheral failures

The three types of conditional command needed to monitor these three types of program event are as follows:

1 IF HALTED

2 IF DELETED

3 IF FAILED

Note: Negative versions of these conditions can be specified; for example: IF NOT HALTED.

The conditional commands can be further qualified to test the program event message. Thus in the example, the user tests whether the program has output the message: PROGA OK on being deleted by means of the command:

IF DELETED "PROGA OK' ,...

Once GEORGE has loaded and entered a program, the program will run until a program event occurs. At this point the program is suspended and GEORGE reads and acts upon the commands following ENTER in the job description until, in the example, the ENDJOB command is obeyed.

THE PRINT COMMAND

If a program fails, the user may wish to know the contents of certain areas of the program in core. The PRINT command can be used to write specified parts of the program area to the monitoring file. It is most likely to be used in conjunction with conditional commands, for example the command:

IF FAILED, PRINT (0,7)

will cause the contents of the accumulators to be output if the program fails.

THE RESUME COMMAND

If he wishes to continue with the object program run after a program event, the user must issue a RESUME command. By including the number of the desired entry point (relative to word 0) the user can specify at which point in the object program he wishes to restart processing. If he wishes to continue from where the program has halted, he issues a RESUME command with no parameters.

THE RUNJOB COMMAND

In the example the use of RUNJOB after IF DELETED . . . illustrates the way in which the initiation of a job can be made dependent on the success of a previous job. If NEXTJOB is initiated, control passes to the commands contained in its job description file. When NEXTJOB has finished, control returns to the command following the one that initiated NEXTJOB, that is, to ENDJOB in MYJOB's job description.

Index

Absolute names 17
Accounting facility 5, 19
Accounting subject program 48
ALLCHAR mode 14, 31
Allowances (budgeting) 47
ALLOWANCE command 48
Amorphous files 14
APPEND command 19
Appending to serial files 14
ASSIGN command 11, 32
ASSOCIATE command 36
ATTRIBUTE command 35
Automatic Operator 6

Background jobs 11
  with MOP 39
Backing Store 3
Back-up system 19
Basic peripherals, simulated 11
Basic peripheral files 13
  modes of access to 14
Batch compilers 6
Batch processing remote 3
Break-in facility 38
Break-in levels 38
BUDGET command 48
Budgeting 47
Budget period 47
Built-in commands 7
Built-in privileges 48

CANTDO command 36
Character strings 21
Clusters, peripheral 36
Command language 2, 7
  in GEORGE 1 and 2 6
Command processor levels 8
  and break-in 38
Commands, built-in 7
  conditional 9
  macro 7
Communications files 14
Compilers 1
  batch 6
  calling in, from MOP terminal 39
Computing Power Index (CPI) 44
Conceptual multiplexers 35
Console property 36
Console typewriters (7021's) 4, 37
CONTINUE command 38
Conversing with an object program 40
Conversion of entrant categories 29
COPYIN command 29

COPYOUT command 29
CREATE command 33
Current directory 17

DEAD command 29
Deleting with the Editor 24
Dictionary 15
Direct access devices, on-lining 34
Direct access files 14
  modes of access to 15
Directories, current 17
  depths of 17
  master 15
  proper 15
DIRECTORY command 18
Disconnecting MOP jobs 11
Dumps, incremental 20
Dump Tape processor 20

Early Morning Start 37
EDIT command 21
Editing language 21
  deleting text 24
  ending the edit 26
  Forget (F) instruction 26
  Insert (I) instruction 25
  Pointer (P) instruction 24
  Replace (R) instruction 26
  Transcribe (T) instruction 23
Editor facility 21
E instruction 26
Elements of multifiles 14
Embedded data 11, 32
Embedded INPUT commands 31
  from MOP terminal 36
Endpoint 22
Entrant concept 13
Entrants in the filestore 13
  outside the filestore 27
ERASE command 33
Erasing files 33
EXECUTE mode 19
Executive, functions of 1
  and multiprogramming 2
Executive scheduling 4, 45
EXPRESS command 44
Exofiles 27-28
Extracodes 1

Fair waiting time 44
FILEIN command 29
Filenames 17
File retrieval system 20

Files, amorphous 14
basic peripheral 13
communications 14
direct access 14
formats of 13-14
job description 2
magnetic tape 14
modes of access to 14
referring to 17
serial 13
stacked 27
terminal 15
Filestore 13
structure of 15
reasons for structure of 18
Forget (F) instruction 26
Fully started jobs 44

General restore 20
GEORGE 1 and 2 5
GEORGE 4 5
GET command 27
GETONLINE 27
GIVE parameter 48
High level scheduler 43

IF command 9
Incremental dumper 20
Increments 20
INPUT command 10, 31
from MOP terminal 36
Input facilities 31
Input and output for object programs 32
Insert (I) instruction 25
INSTPARA command 37

JOB command 10
Job description: GEORGE 1 and 2 6
GEORGE 3 and 4 2, 11
file 2
once-only 10
permanently stored 10
JOBLIMIT parameter 37
Jobs, background 11
background with MOP 39
fully started 44
on-line (MOP) 4, 11
tentatively started 44
JOBTIME command 44
Juggernaut restorer 20

Labels 7
Language, command 2, 7
editing 21
in GEORGE 1 and 2 6
Levels, command processor 8
break-in 38
Librarian 27
LINGO(fictional) macro 8
LISTFILE command 11, 20, 32
and peripheral clusters 36
and properties 35
LOAD command 20

Local name 17
Log analysis program 5
Logging 5
LOGIN command 37
Low level scheduler 44

Macro commands 7
nested macros 8
system macros 8
user macros 8
Macro levels 8
Magnetic tape files 14
modes of access to 15
on-lining 34
MAKEDIR command 15, 47
MAXSIZE command 44
MINIMOP (GEORGE 1 and 2) 6
MONEY budget 47
Monitoring file 10
MOP 4, 37
MOP terminals as input/output devices 36
Multifiles 14
Multiplexers 35
Multiprogramming with Executive 2
with GEORGE 3 and 4 2

Nested macros 8
NEW command 29
NEWUSER privilege 47
NORMAL mode 14, 31

Off-line peripherals 32
Off-lining with GEORGE 2 6
with GEORGE 3 and 4 3, 10, 32
ONLINE command 27, 34
and MOP 36
and multiplexers 34
and peripheral clusters 36
and properties 35
On-line job 4, 11
On-line peripherals 34
Operating systems, objectives 1
facilities: GEORGE 3 and 4 2
other operating systems 6
Operator, functions of under Executive 1
under GEORGE 3 36
Output from object programs 10, 11, 32

Paged 1900's 5
Paper tape input 31
Parameters 7
blocks 9
identifiers 9
in user macros 8
locations 9
Passwords 37
Peripheral clusters 36
Peripheral names 34
Peripherals, basic (on-lining) 34
PERIs 13, 14
Pointer 22
Pointer (P) instruction 24
Pool tapes 27

Privacy 19
Privileges 48
Program development with MOP 4
Proper directory 15
PROPERTY command 35
Property consoles 36
Property names 35
Property system 34
PUC (Program under control) 5
Pure code 5

Quit (Q) instruction 26

Rations (in budgeting) 47
READ access modes 19
REALTIME budget 47
Relative names 17
Remote batch processing 3
Remote data terminals (7020's) 3, 35
RENAME command 10
Replacing (R) instruction 26
REPORT command 38
RETRIEVE command 20

Secure entrants 27
SETPARAM command 9
Shared programs 5
Simulated basic peripherals 11
SPACEMT budget 47
Stable budgets 47
System macros 8

TAKE parameter 48
Teletypewriters (7023's) 35
Tentatively started jobs 44
TERMINATE command 36
TIME budget 47
TIME command 39
Time slot 44
TRACE command 10
Transcribe (T) instruction 23
Transient budgets 47
TRAPGO command 19
TRUSTED privilege 40
Typewriters, console (7021's) 37

URGENCY command 44
User macros 8
User names 17
User, proper 17
User traps 19

Verbs 7
Virtual store (Paged 1900's) 5

WHATPER command 36
WHENEVER command 9
Workfiles 33
Working job description file 10
Worktapes 27
WRITE access mode 19