

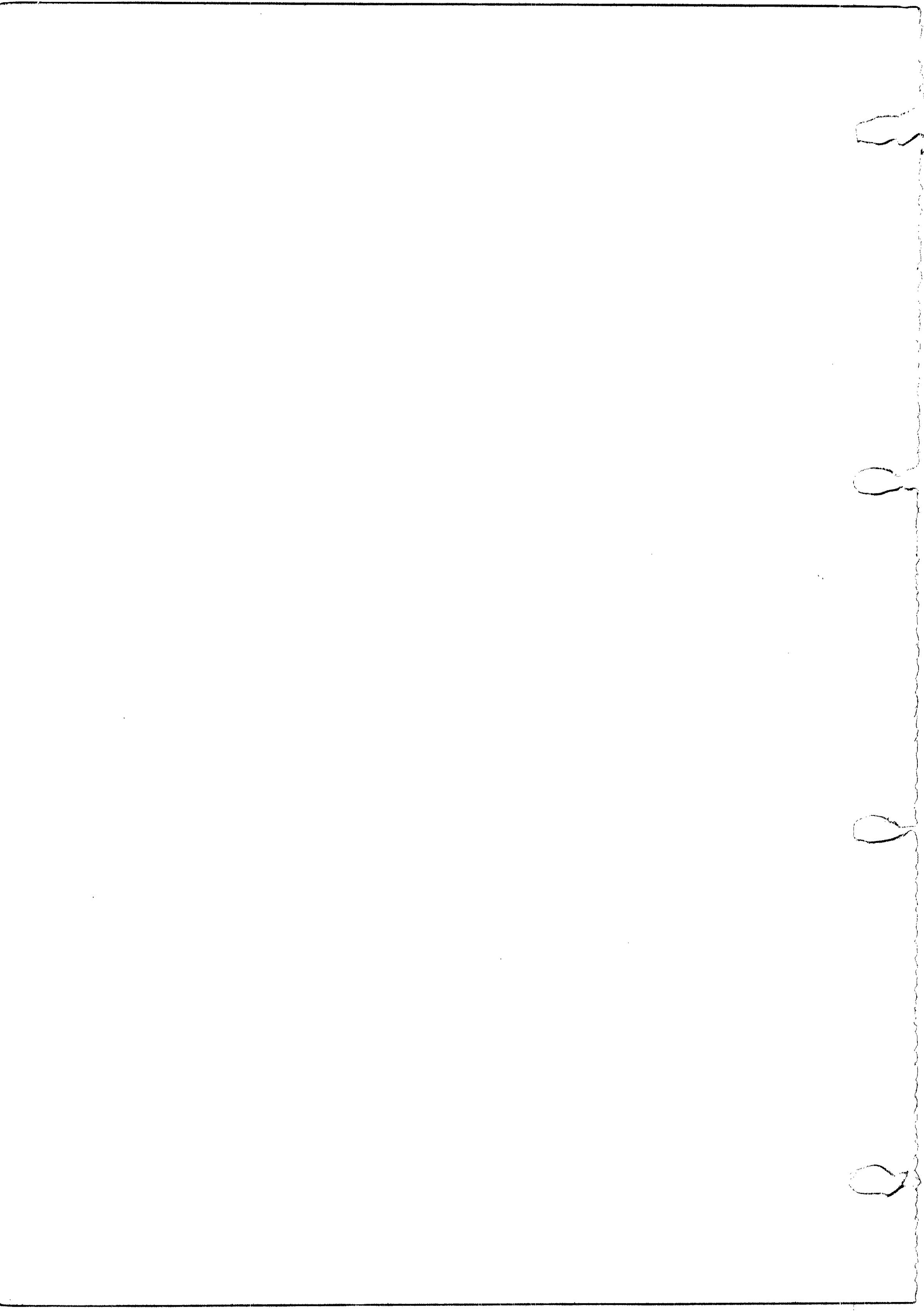
PREFACE

The Algol 68-R system is a program compilation and execution system based on a compiler for ALGOL 68. A readable account of the ALGOL 68 language is given in the booklet "Algol 68-R Users Guide" published by Her Majesty's Stationery Office. This "Programmers Manual" contains further information needed during actual programming work. It includes a full description of input and output facilities and the specification of the Algol 68-R system library. Details are also given of program segmentation and the "album" facility.

The manual is reproduced from the general manual of the RRE Computing Service. It thus describes the use of Algol 68-R under the GEORGE 3 Mark 8 Operating System. Chapters dealing with standard George commands have been omitted. There are minor differences when Algol 68-R is used with Operators Executive or GEORGE 2, and these are noted in the booklet "Installation and Maintenance" issued with the system software.

*Distributed by Software Distribution Department,
International Computers Ltd*

sgb(2.76)



NOTATION

In extracts or examples of programs, job descriptions, etc., and in explanations of their form, the following conventions are as far as possible adopted in the Manual.

1. An underlined word indicates the form of a construction.
For instance,

FORGET segname

means the actual word FORGET followed by a "segname" whose particular form must be sought from the accompanying text, or failing that, from the Index.

2. Dashed underlining has the same meaning as full underlining, except that the item may optionally be omitted. For instance,

fileidentifier details

indicates that "details" may be altogether omitted.

3. Algol 68-R programs punched on paper tape using the normal character set make use of upper and lower case letters, and it is essential to use the two cases correctly. Job descriptions punched on paper tape can be punched in either case, as George makes no distinction. For ease of reading, however, it is a convention in the Manual to use upper case for language words such as George "verbs" and lower case for identifiers and similar words. For instance,

TRAPGO pmw, ALL

includes the George words TRAPGO and ALL, whilst pmw is the name of a file which is made up arbitrarily by a user.

0

C

O

0

GEORGE

Contents

*1 INTRODUCTION

- 1.1 Preamble on files and users
- 1.2 Principal George commands
- 1.3 Format of a George command
- 1.4 George identifiers
- 1.5 DISENGAGE

2 FIRST NEEDS IN JOB CONTROL

- 2.1 Compiling and executing a new program
- 2.2 Limits on simple jobs
- 2.3 Filing data separately
- 2.4 Filing a program using upper and lower case alphabets
- 2.5 Jobs with larger limits
- 2.6 Example of a simple job and its output

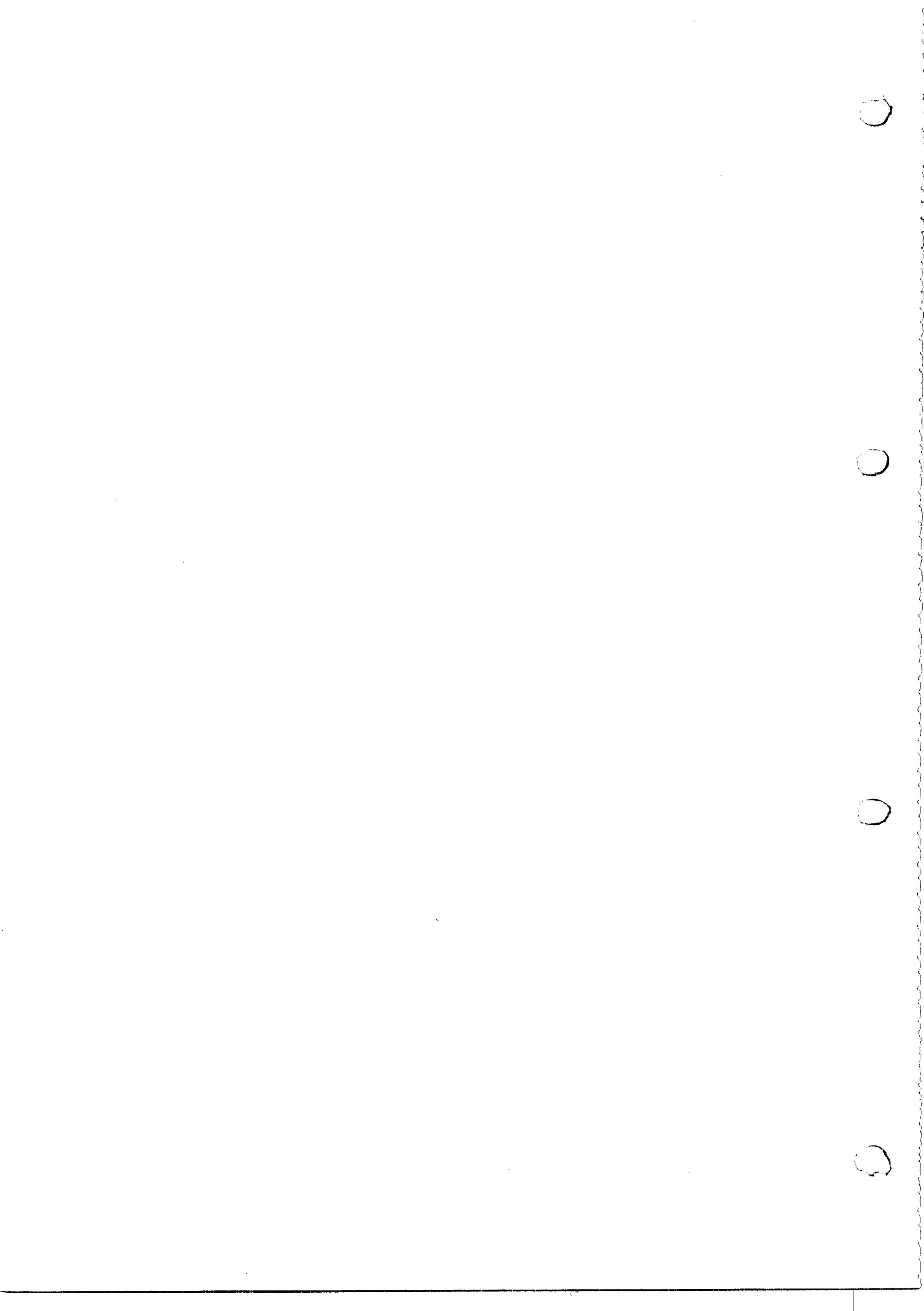
*3 THE JOB

- 3.1 The JOB command
- 3.2 The monitoring file system
- 3.3 ENDJOB
 - 3.3.1 Keeping a copy of the monitoring file
- 3.4 JOBTIME
- 3.5 Errors in commands - WHENEVER
- 3.6 Obeying a filed job description - RUNJOB

4 COMPILING AND RUNNING

- 4.1 The COMPILE - EXECUTE sequence
 - 4.1.1 Running without recompilation
 - 4.1.2 Compiling only
 - 4.1.3 Large programs
 - 4.1.3.1 Finding out the size of a program
 - 4.1.4 Running in Extended Branch Mode
 - 4.1.5 Overlaying
 - 4.1.5.1 Overlay patterns for a single overlay area
 - 4.1.5.2 Use of more than one overlay area
 - 4.1.5.3 Layout of an overlay specification
 - 4.1.5.4 Designing an overlay specification
 - 4.1.5.5 Overlaying in Extended Branch Mode
- 4.2 Commands between COMPILE and EXECUTE
 - 4.2.1 CHECK
 - 4.2.1.1 CHECK INDEX
 - 4.2.1.2 CHECK OVERFLOW
 - 4.2.2 TIME
 - 4.2.3 STORE
 - 4.2.4 Connecting input data files to the program
 - 4.2.5 Output from the program
 - 4.2.5.1 Putting output into a file
 - 4.2.5.2 Adding output to a file, or producing more than 500 lines
 - 4.2.6 Connecting binary files to the program
 - 4.2.7 Using switches
- 4.3 Programs which run for more than 30 minutes
 - 4.3.1 The "save" library procedure
 - 4.3.1.1 Saving a program which has graphical output
 - 4.3.2 Job descriptions for long running programs
 - 4.3.3 Saving an overlaid program

* Issued to RPE users only



2 FIRST NEEDS IN JOB CONTROL

2.1 Compiling and executing a new program

In simple cases, prepare a single paper tape or deck of cards according to the following pattern. On paper tape, leave a foot or so of blank before the perforation starts. Underlined words are symbolic - they must be replaced by whatever applies in your case. A filename is any word of 12 characters or less, the characters being letters, digits or hyphens; the same applies for a jobname. Do not include spaces within a name. It is good practice to start each name with personal initials, eg jbs-testprog. Unless otherwise instructed George does not distinguish between upper and lower case alphabets. In the program enclose language words in primes, and on paper tape do not use the symbols { } or |. The numbers on the left are for reference in this explanation only.

```
1 INPUT :user, filename
2 program
3 ****
4 JOB jobname, :user
5 WHENEVER COMMAND ERROR, ENDJOB ALL
6 COMPILE filename
7 CHECK INDEX, OVERFLOW
8 EXECUTE
9 data (maximum 5 lines)
10 ENDJOB
11 ****
12 DISENGAGE
```

Begin each of the above items on a new line or card, and when using paper tape finish off by punching carriage return, line feed, after item 12.

Notes

- 1 The input command to George. All information following this line, up to and including ****, will be filed. The username (colon and name) is allocated by the computer manager for your work. The file name is the name of the file into which items 2 and 3 will be written. If the name is a new one, a new file will be created and will remain in the system holding the information indefinitely (unless deliberately removed). If the file has been created before, its previous contents will be replaced by items 2 and 3.
- 2 Algol 68-R program, starting with program title in the form of an Algol identifier and ending with the word 'FINISH'. See Algol 68-R Users Guide. To use the full character set with upper and lower case alphabets see 2.4.
- 3 Terminates the information to be filed (and is itself filed). George now expects another command.
- 4 The job command to George. The name of the job exists for the duration of the job and is then forgotten. It must be distinct from any file name existing under the given username. George now expects a sequence of commands describing the job to be done.
- 5 A necessary safety precaution.
- 6 Compiles the Algol 68-R program from the file named. The name must therefore be that given in item 1.
- 7 An essential command when testing a new program, and advisable for all programs which run for less than 1 minute. Causes checking of array subscripts and arithmetic overflow during running of the program.
- 8 Causes program compiled at 6 to be executed.
- 9 Algol 68-R "read" procedure takes data from here, unless the data is filed separately. Data should be taken from its own file if it is more than a few lines long (see 2.3).
- 10 Tells George that the end of the job has been reached.
- 11 Acts as a terminator for the preliminary filing of the job description (items 4-11 inclusive).
- 12 Stops the reader; no further cards or information on the tape will be read. Essential to the computer operator. On paper tape the word DISENGAGE (or DG for short) must be terminated by a carriage return, line feed.

2.2 Limits on simple jobs

The following limits are imposed on programs controlled as described in section 2.1.

Program working space 1500 words

A program needs 500 words for working space plus space for any arrays declared. The array space needed can be estimated from the size of the array and its elements.

Integers take 1 word each

Reals " 2 words each

Complex " 4 words each

Output 500 lines (about 8 pages)

Time of running 1 minute (ample for testing)

Time for complete JOB 5 minutes (sufficient for compilation and a 4-minute run)

To overcome these limits, see section 2.5.

2.3 Filing data separately

Data of any sort can be filed in the same way as a program. Between items 3 and 4 of the box in section 2.1, the following insertion can be made:

```
INPUT :user, filename
data
****
```

Alternatively, this could be a separate tape or deck of cards, ending with DISENGAGE (and on tape, carriage return, line feed).

To make the Algol 68-R "read" procedure take its data from a separate file instead of position 9 in the box of 2.1, the command

```
ASSIGN *tr0, filename
```

must be inserted anywhere between the COMPILE and EXECUTE commands, as illustrated in 2.5. The 0 is a zero.

2.4 Filing a program using upper and lower case alphabets

If a program is typed on paper tape using the full character set described in the Algol 68-R Users Guide, item 1 of the box in 2.1 must be replaced by

```
INPUT :user, filename (NORMAL)
```

2.5 Jobs with larger limits

The following job description should be compared with lines 4 to 10 of the box in 2.1.

```
14   JOB jobname, :user
15   WHENEVER COMMAND ERROR, ENDJOB ALL
16   JOBTIME 6 MINS
17   COMPILE filename
18   TIME 5 MINS
19   STORE 2000
20   LIST LIMIT 1000
21   ASSIGN *tr0, filename
22   EXECUTE
23   ENDJOB
```

- Notes
- 16 Limits time for the complete job to 6 minutes instead of 5.
 - 18 Limits execution time of program to 5 minutes instead of 1. This will give the job a lower priority with the computer operator, and should be used with discretion.
 - 19 Limits program working space to 2000 words instead of 1500. A good working rule for the figures is to estimate the space required for arrays and add 500 words.
 - 20 Limits printed output to 1000 lines instead of 500.
 - 21 Allows for any amount of separately filed data, see section 2.3.

Note that the command 7 in section 2.1 can be omitted for a thoroughly tested program. This will result in more rapid (and riskier) execution. Any of the commands 18-21 which are required can be used as shown and can be placed in any order, provided that they appear between COMPILE and EXECUTE.

2.6 Example of a simple job and its output

[This example uses the full character set for the program and thus needs the extra bracketed word NORMAL enclosed in round brackets as described in section 2.4].

```
INPUT :manuals, pmw-ex(NORMAL)
squaring
BEGIN REAL x := 1.0;
    print(("      X          X↑2", newline));
    WHILE x > 0.0 DO
        BEGIN read(x);
            print({newline, x, "      ", x*x})
        END;
        print(newpage)
    END
FINISH
*****
JOB pmw-ex-run, :manuals
WHENEVER COMMAND ERROR, ENDJOB ALL
COMPILE pmw-ex
EXECUTE
3.1415927
1024
-1
ENDJOB
*****
DISENGAGE
```

First output document - results from program

#OUTPUT BY LISTFILE IN ':MANUALS.PMW-EX-RUN' ON 6OCT71 AT 13.23.16

[new page]

X	X ²
+3.1415927000& +0	+9.8696046927& +0
+1.02400000000& +3	+1.0485760000& +6
-1.00000000000& +0	+1.00000000000& +0

Second output document - monitoring information

A monitoring printout is supplied with each completed job. It shows the progress of the job by giving each command which was obeyed by George, with extra comments and messages as necessary. The exact format and the type of information given may vary from time to time, but the following extract shows the significant lines to look for. Since the monitoring output is your only guide to what actually happened in a job it is essential to read it. It may also be vital for the planning of future jobs, and should not be thoughtlessly destroyed.

#OUTPUT BY LISTFILE IN ':MANUALS.PMW-EX-RUN' ON 6OCT71 AT 13.23.25

DOCUMENT :MANUALS.PMW-EX-RUN(/B1B0)

STARTED :MANUALS,PMW-EX-RUN, 6OCT71 13.15.50
13.15.50< JOB PMW-EX-RUN, :MANUALS

[Time in hours, minutes and seconds,
when JOB command was obeyed.

13.15.52< WHENEVER COMMAND ERROR, ENDJOB ALL
13.15.52< COMPILE PMW-EX

0.04 :HALTED : COMPILED SQUARING

[0.04 is the cumulative total of charged
time spent on the job, in minutes and
seconds. The compilation was success-
ful; had it failed, compilation error
messages would appear here.

13.22.08< EXECUTE

13.22.49 0.06 CORE GIVEN 4992

[6 seconds used, 4992 words of space
required for program plus data.

DISPLAY : LD

[Program loaded, ready to go.

0.08 :DELETED : OK

13.22.57 0.08 DELETED,CLOCKED 0.01

[Program ran faultlessly; 8 seconds used
on job so far, of which 2 seconds were for
the program run (0.08-0.06). This
information enables TIME needed for longer
runs to be estimated (to the nearest
minute). The figures after CLOCKED should
be disregarded.

13.22.59< ENDJOB

13.22.59 0.08 FINISHED : 2 LISTFILES

[8 seconds of JOBTIME used altogether; the
complete job produced 2 output documents -
the results from the program and this
monitoring file.

4 COMPILING AND RUNNING

A program cannot be obeyed until it has passed through the compiler which translates it from the source language Algol 68-R into the object code required for actual running. The unit of source text submitted for compilation at any one time is called a segment; simple programs are usually written in one segment and compiled and run using the sequence of commands described in sections 4.1 and 4.2. Variations for compiling only or running only are described in sections 4.1.1 and 4.1.2. Details of how to divide a program into pieces for separate compilation are given under Albums in the UTILITIES section of the manual.

4.1 THE COMPILE - EXECUTE SEQUENCE

The text of a segment to be compiled must first have been put into a file (usually by a separate INPUT command). The command to compile uses the name of this file (*not the title of the segment*), thus

COMPILE filename

If the segment compiles correctly the message

HALTED: COMPILED segmentname

goes to the monitoring file. Otherwise diagnostic information follows, and the whole job is ended.

If the compilation has succeeded, the segment can be obeyed by the EXECUTE command, though some additional commands will often be needed between compilation and execution. The sequence is

COMPILE filename

auxiliary commands, if any

EXECUTE

The auxiliary commands affect the way the program is run and are described in section 4.2. Data required by the procedure "read" inside the program can be punched after EXECUTE, starting on the next line or card. For reasons of efficiency, such data should not extend over more than about 5 lines. Care should be taken that all the data is actually read by the program, as any left over at the end of a run will be treated by George as the next command in the job description. Data in large quantities should be filed separately, and the data files connected to the program as described in section 4.2.4.

If the program fails during execution, run-time diagnostics go to the monitoring file and the whole job is ended. Note that as the EXECUTE command has no parameters it can only be used to run a program after the name of the final (or only) segment of the program has been made known to it, in this case by the COMPILE command. The EXECUTE command can thus only be used in a COMPILE - EXECUTE sequence or in a loading and EXECUTE sequence as described in sections 4.1.1, 4.1.4 and 4.1.5.

4.1.1 Running without recompilation

Whenever a segment is compiled, its compiled form is put into a system file and held there under its segment name (- not the name of a file, but the Algol identifier at the beginning of the actual text) until the space it occupies is needed for further compilations. It is thus available for a time depending on the amount of work being done by the computer, and can be recovered and rerun later in the same job by the commands

GETCOMPILED segmentname
auxiliary commands
EXECUTE

The GETCOMPILED command is used instead of COMPILE and the phrase 'between COMPILE and EXECUTE', wherever it appears in this manual, means alternatively 'between GETCOMPILED and EXECUTE'. If the segment is no longer available the message SEGMENT NAME NOT FOUND will appear on the monitoring file, and the source text will have to be recompiled. To keep a compiled segment permanently for repeated running it must be put into an album after compilation, as described in UTILITIES Albums. It can then be recovered and run by the sequence

GETCOMPILED segmentname, albumname
auxiliary commands
EXECUTE

* 4.1.2 Compiling only

As well as arranging for compilation, COMPILE does some of the preparation for program running needed before the EXECUTE command can continue. This preparation is unnecessary if EXECUTE is not to follow, and can be eliminated by using

COMPILEONLY filename

which merely arranges for compilation. This command is particularly useful on a MOP terminal as it does less work and thus gives a slightly better response. Common situations in which COMPILEONLY can be used are:

- 1 When the compilation is to be followed by UPDATEALBUM to move the compiled segment into an album.
- 2 When a program is large and needs special loading as described in section 4.1.3.
- 3 When the compilation is only for checking the source text.

4.1.3 Large programs

When a program is run by EXECUTE, the necessary code is assembled and loaded into store. For this to be accomplished successfully, space must have been found for a number of compiled segments. Library segments will be present, in response to the use made of library facilities, and previously compiled segments of the user's own program will also have been assembled in front of the final main segment. Space also has to be found for extra code needed for run-time checking as specified by the auxiliary command CHECK (see 4.2.1). For a large program, these space requirements may prove to be too great for the hardware limits of the machine, and the loading process then fails. There are two types of limit, one causing the message LOWER EXCEEDS 4096, and the other giving PROGRAM TOO LARGE FOR NORMAL BRANCH MODE.

LOWER EXCEEDS 4096 means that the 'lower store' limit of 4096 words has been reached. 'Lower store' is a 1900 hardware feature, and in Algol 68-R programs it is used for storing the globally declared objects of all the segments of a program (excluding arrays) together with the sequences of characters which appear in any format, bytes or string denotations written in the program. Exceeding the limit is normally caused by having a large number of denotations rather than too many globally declared objects, and before the program can be successfully loaded and run it must be rewritten with fewer of these (for example, strings could be input to the program rather than written as denotations). The amount of 'lower' used is given by SEGMENTSIZES, described in the next section, but note that SEGMENTSIZES will itself fail if the limit is exceeded.

PROGRAM TOO LARGE FOR NORMAL BRANCH MODE means that the number of machine code instructions in the program together with the amount of 'lower' used is greater than 32768. If this happens, the SEGMENTSIZES of a version of the program which will load successfully should be obtained as described in the next section, and the program should be reviewed briefly. There are two possible courses of action. Firstly, the program can be loaded as described in 4.1.4 to run in a special hardware mode known as Extended Branch Mode (EBM), which has no 32K limit. Alternatively, the overlay loader can be used as described in section 4.1.5. This has the advantage of reducing the core occupied by the instructions of the program, thus reducing its total core requirement, or allowing more data space to be requested with no increase in the requirement.

4.1.3.1 Finding out the size of a program

A complete list of the segments constituting a program and their sizes, in the order in which they are assembled, can be obtained by using the command SEGMENTSIZES. This command is given instead of EXECUTE, that is, it must appear after COMPILE or after one of the commands which retrieve a previously compiled segment and load a program in some way, for example

```
GETCOMPILED segmentname, albumname
CHECK OVERFLOW
SEGMENTSIZES
```

The only auxiliary command needed is CHECK and this must be given exactly as it would be for the program run, since the amount of checking applied affects the sizes. The output from SEGMENTSIZES appears on the monitoring file of the job. The name of each segment used is given, together with its size in words; some segments from the system library are specially marked for those users deciding on overlay specifications (described in 4.1.5). The first item in the output ('LOWER') is only of interest if its size approaches 4096, as described in the previous section. The total core required to run a program is the amount from SEGMENTSIZES together with the data space requested by STORE (section 4.2.3), and some extra introduced by the system.

4.1.4 Running in Extended Branch Mode

A program can be loaded and run in Extended Branch Mode (EBM - described in section 4.1.3) by using the command sequence

```
GETEBM segmentname, albumname
auxiliary commands
EXECUTE
```

that is, GETEBM is used in the same way as GETCOMPILED (which loads the program normally). Note that there is no COMPILE - EXECUTE sequence for running in EBM - the program must first be compiled and then explicitly retrieved for immediate running, or put into an album and recovered from there. The phrase 'between COMPILE and EXECUTE', wherever it appears in this manual, applies equally between GETEBM and EXECUTE.

4.1.5 Overlaying

It is possible to reduce the total core store required for a program run by holding a substantial proportion of its instructions on backing store when they are not in immediate use. A special loader, known as the 'overlay loader', can be used to organize the necessary transfers of program into and out of main store, which will take place during the course of the run. The smallest unit of program which can be moved about in this way is a *segment*. Inevitably overlaying will increase the running time of a program, to an extent dependent on the pattern of segmentation and overlaying adopted. This is under the user's own control and should be worked out with some care, as described in later sections. The specification should then be put into a file for use as data by the overlay loader, which is brought into play by the command sequence

GETOVERLAY filename

auxiliary commands, as between GETCOMPILED and EXECUTE

EXECUTE

The command GETOVERLAY is a substitute for GETCOMPILED, and the file filename must contain the required overlay specification in a form to be described. (Alternatively, GETOVERLAY can be used with no parameter at all, and the overlay specification included in the job description itself, starting on a new line after the GETOVERLAY command.)

The overlay specification has two parts. The first line gives the information normally required by a GETCOMPILED command, ie the name of the main segment and possibly an album name, as described below. The second and any subsequent lines give the actual pattern for the overlays. For the moment, consider only the first line. If a main segment entitled "radarprog" (say) has just been compiled, the first line is simply "radarprog", thus

Command sequence

GETOVERLAY filename
auxiliary commands
EXECUTE

Contents of the named file

radarprog
overlay pattern, possibly extending over more than one line

If on the other hand "radarprog" had been previously compiled and put into "radalbum" (say), the first line would be "radarprog-radalbum". If the album does not belong to the user running the program, its name must be specified in full (:user.filename). For example,

radarprog - :sigs.radalbum
overlay pattern, possibly extending over more than one line

The four stars shown merely indicate that the specification is normally kept in a file; they are not part of the specification, and are not used or needed by the overlay loader. Note the hyphen separator between the segment name and the album name.

4.1.5.1 Overlay patterns for a single overlay area

Consider a program whose main segment is headed

radarprog WITH seg1, seg2, seg3 FROM myalbum

This would seem to indicate a program having four segments in all, but it should not be forgotten that there may be other consequential segments from the album and almost certainly library segments in addition. All of these can also be brought into the overlay pattern if desired, with some restrictions (noted in 4.1.5.4).

Suppose, for simplicity, that it is decided to have either seg1 or seg2 in main store at all times, but never both simultaneously. The overlay pattern would then be typed as

seg1, seg2.

terminated by the full stop, as shown. This specifies that all segments not mentioned (including library segments) shall be present in main store the whole time, but that seg1 and seg2 shall be treated as alternatives, ie swapped in and out. The overlay loader will set aside a single area in main store of suitable size for this purpose. The overlay specification as a whole would be filed as

radarprog - myalbum [Assuming program in album
seg1, seg2. [actual overlay pattern

Any number of segments can be made to be alternatives in the one overlay area. For example,

seg1, seg3, radarprog.

would mean that main store would hold the library segments and seg2 permanently, and the overlay area would hold either seg1, seg3 or the main segment radarprog.

Very short segments are not good candidates for overlaying, as little space will be saved. A moment's reflection will show that for two alternative segments, the saving of space in main store cannot be more than that occupied by the smaller of the two. It may then be sensible to transfer several segments at a time, treating them as a single unit. This can be represented in the overlay pattern by using plus signs. For example,

seg1, seg2 + seg3.

will make the overlay area hold either seg1, or seg2 together with seg 3.

4.1.5.2 Use of more than one overlay area

An overlay pattern can specify more than one overlay area. The pattern for each area is written as already described, with semi-colons as separators between areas and full stop termination for the whole sequence. For example, consider

a+b, c; d, e+f, g.

This sets aside two areas, and allows six different combinations of segments in store.

For even greater flexibility, it is possible to arrange for an area to be subdivided dynamically in the course of a run. As an example,

p, q, (r+s, t; u, v).

sets aside an overlay area to hold p or q, or to be subdivided into two areas. When neither p nor q is in main store, one of r+s and t, and one of u and v will be resident. A bracketed pattern can always be used as an alternative for an area.

4.1.5.3 Layout of an overlay specification

The first line of an overlay specification may not contain more than 127 characters. Subsequent lines of the specification, which give the actual overlay pattern, are also limited to 127 characters but layout is ignored, so spaces and new lines can be taken wherever required (but not in the middle of a segment name). The overlay pattern as a whole must always be terminated by a full stop as described in the previous sections.

4.1.5.4 Designing an overlay specification

The principles to be used in deciding on a pattern of overlays are

- (1) Economy in use of main store
- (2) Avoidance of excessive backing store transfers

To satisfy (1), the sizes of all the segments of a program should first be ascertained by use of the command SEGMENTSIZES (4.1.3.1). Some segments from the system library cannot be overlaid, but these are clearly marked in the output from SEGMENTSIZES. No segment may appear twice in an overlay pattern, and all segments not mentioned will be resident in main store throughout. The message SEGMENT NAME NOT FOUND when attempting to load an overlaid program means that a segment mentioned in the overlay pattern is not one of the segments of the program, or has already been mentioned, or is a library segment which cannot be overlaid. When a scheme has been arrived at, the final store allocation should be inspected; this again is done by using the command SEGMENTSIZES in place of EXECUTE, but now with the overlaid program, ie by the command sequence

GETOVERLAY filename
auxiliary commands
SEGMENTSIZES

To satisfy the second principle (avoidance of excessive backing store transfers), commonsense is called for. For example, if a procedure is declared in one segment and called in another, it should be borne in mind that every call of the procedure would involve two backing store transfers unless both segments were in store simultaneously. Each transfer may take at least a quarter of a second. To guard against an excessive number of transfers, the monitoring file for any run of an overlaid program should always be studied. At some point after DELETED OK or DELETED ER, the message

FREE *DA13, integer TRANSFERS

will be found. The integer gives the number of times an overlaid segment was not available in main store. If this number is large in relation to the time taken by the program, it is a strong indication that the overlay specification is unsuitable.

4.1.5.5 Overlaying in Extended Branch Mode

Occasionally, with extremely large programs, it may be necessary to run in Extended Branch Mode (EBM), and also to use overlaying. The command sequence to do this is

GETOVEREBM filename

auxiliary commands, as between GETCOMPILED and EXECUTE

EXECUTE

that is, GETOVEREBM is used instead of GETOVERLAY, and in exactly the same way.

4.2 Commands between COMPILE and EXECUTE

The auxiliary commands between COMPILE and EXECUTE affect the way the program will be set up to run. Every input channel in the program must be connected to its source of data, unless there is only one channel and the data is embedded in the job description itself (after EXECUTE). Every output channel must also be dealt with, unless there is only one channel and the output is less than 500 lines. The time to be allowed for the run must be given, unless it is less than one minute, and similarly the amount of data space needed must be estimated. Also between COMPILE and EXECUTE a command to introduce extra run-time checking can be given - this is essential for programs undergoing test.

4.2.1 CHECK

This command is used for two purposes, to provide a run-time check on array indexes and on arithmetic overflow. When the full form of the command is used, ie

CHECK INDEX, OVERFLOW

both forms of checking will be applied. This is also the case if CHECK is given on its own, without parameters. The use of CHECK slows down the running of the program, but is strongly advised for program testing, and for short-running programs (say less than 1 minute).

4.2.1.1 CHECK INDEX

Checks that array indexes are used within their bounds. An indexing error causes the message

INDEX OUT OF BOUNDS

to be displayed, while a slicing error gives rise to the message

SLICE TOO LARGE

4.2.1.2 CHECK OVERFLOW

When the numerical result of a calculation is larger than the machine can store (approximately 5.8×10^{76} for reals and 8388608 for integers), a register is set and rubbish is stored. It is important to realize that the machine does not automatically report overflows and will carry on computing with rubbish until the overflow register is explicitly checked. Checking always takes place on entry to a library procedure; if overflow is found to be set, the message

OVERFLOWED AT CALL OF LIBRARY PROC

will be displayed, and the program will enter the diagnostic routine. There is one exception to checking on entry to library procedures; with "print", the report does not occur until after printing has been carried out.

The checking described above is far from adequate, and is augmented by the CHECK OVERFLOW command. This inserts an extra check after every explicit assignment (ie those caused by := but not by the assignment operators PLUS etc) and whenever an actual parameter is evaluated (which includes evaluations of right-hand sides of identity declarations). When overflow is detected, the message

OVERFLOW SET

is displayed and the diagnostic routine entered. Unfortunately, there are still loop-holes which enable overflows to pass completely undetected. For example, when an integer is converted to a real, the overflow register is cleared by the hardware, and the capability of testing for overflow is lost. It is instructive to compare the three assignments in the following extract from an Algol 68-R program.

```
REAL x := 1.0&9;  
REAL y := 10.0↑9;  
REAL z := 10↑9;
```

In the first, "1.0&9" is a shorthand notation for 1000000000, which is what the compiler understands by it. The second assignment is bad programming, because it is effectively a formula, requiring 10.0 to be raised to the power 9 at run-time. It is always wasteful to use operators unnecessarily, like writing 3+4 instead of 7. The third assignment, also bad programming, actually causes an undetected overflow. The formula 10↑9 is evaluated as an integer, sets overflow, is then converted to real and overflow is cancelled. The number then assigned to z happens to be 10144256.0, and no alarm has been sounded.

4.2.2 TIME

Programs are not allowed to run for longer than one minute unless a TIME command is included between the COMPILE and EXECUTE commands. The alternative forms are

TIME integer MINS

TIME integer SECS

whilst if MINS or SECS is omitted, seconds are understood. It is advisable to be as accurate as possible when estimating TIME, for if the needs are overestimated, a slower turn-round will result. Guidance can be obtained from the monitoring information supplied with previous runs of the same or a similar program, since this shows the time used at various stages in a job (see GEORGE 2.6).

The TIME command applies to the running time of a specific program. If a run longer than 4 minutes is required, there may be some danger that the total job time will exceed 5 minutes, in which case a JOBTIME command will be necessary (not between COMPILE and EXECUTE), see section 3.4. Long program runs (more than about 30 minutes) should include provision for restarting after a system failure, as described in 4.3.

4.2.3 STORE

Programs are automatically given 1500 words of data space. For more data than this, a STORE command must be included between the COMPILE and EXECUTE commands, thus

STORE integer

causes integer words of data space to be allowed. To estimate data space for simple programs, a useful rule of thumb is to take the space requirements for arrays and add 500. The space for an array is determined by the size of the array and the mode of its elements. Items of mode INT, BOOL, BYTES, BITS, and all references, take 1 word each; items of mode REAL take 2 words and COMPLEX items take 4. LONG in front of a mode doubles the space required; LONG LONG trebles it. An array of characters (elements of mode CHAR) takes 1 word per 4 characters, the space being rounded up to a whole number of words. In addition, there are 5 system overhead words for each one-dimensional array, with 3 more overhead words for each extra dimension. (For example, a 2-dimensional array takes 8 words in addition to the space for its elements.) This overhead applies to all arrays, but may only be significant for character arrays (including STRINGS), which tend to have relatively few elements.

If a program uses heap storage, space for the heap must be included in the STORE allowed. An estimate should be made of the maximum amount of space taken up, and accessible, at any one time by generators (local and global) and arrays. All arrays should be counted, whether generated or declared, and flexible arrays (and hence STRINGS) should also be included. The estimate should be doubled to give an initial integer for the STORE command. When the program runs, it may fault with the message SCAVENGE c - NO SPACE LEFT from the garbage collection process, where c is a character whose ABS value gives the number of times garbage collection has occurred. In this case the STORE should be increased for the next run, after checking that the program is not retaining generated space unnecessarily (for example, by holding references to unwanted space, see Algol 68-R Users Guide, Chapter 10, page 63). If a program using the heap takes more time to run than expected, this may show that garbage collection is occurring frequently, but recovering only a small amount of space each time. Increasing the STORE allowed will reduce the number of garbage collections and decrease the program running time.

4.2.4 Connecting input data files to the program

For each CHARPUT channel opened in the program as a "file reader" the command

ASSIGN *fr channelnumber, filename

must be included between the COMPILE and EXECUTE commands. The channelnumber, equal to or greater than 1, is the one associated with the charput variable by the procedure "openc" (Library, Input/Output 2) and reading procedures using that charput variable will then take data from the file called filename. The data must already have been put into the file - in most cases this would be by a George INPUT command, although it is also possible to use a file generated by output from a previous program.

The procedure "read" is by default associated with a special channel numbered zero. Channel zero normally takes data from the job description (4.1.1) but can be switched to take it from a file by the command

ASSIGN *tr0, filename

The 0 in this command is a zero. The letters tr stand for tape reader, and the same command (with the required channel number instead of 0) is used for any special channel opened in the program as a "tape reader" (see LIBRARY Input/Output 2.5).

Any number of *different* reading channels may be attached to the *same* file. Each channel will remember its position in the file independently of the others.

cts(4.74)

4.2.5 Output from the program

Results from the procedure "print" are automatically produced on a separate line printer document, but if more than 500 lines are required the command

LIST LIMIT integer

must be used between COMPILE and EXECUTE and the effect is to increase the limit from 500 to integer lines.

If there are only a few lines of output from "print" they may be sent to the monitoring file of the job instead of having a separate document. This is done by giving the command

ASSIGN *1p0

between COMPILE and EXECUTE. The 0 in this command is a zero, and the letters lp stand for line printer. The word LIMIT does not apply in this case.

Results from each extra output channel opened in the program can either be printed on the line-printer or put into a file for future use (for example, for the production of paper tape). If a printed document only is required the second parameter of the procedure "openc" for the channel should be "file printer" and the command

LIST channelnumber, LIMIT integer

given between COMPILE and EXECUTE, where channelnumber is the integer used in the procedure "openc" (Library, Input/Output 2). The LIMIT parameter allows up to integer lines of output on the specified channel. If it is omitted, the allowance is 500 lines.

Irrespective of any LIMITS given, there is an absolute maximum of about 10000 lines (approximately 200 line-printer pages) allowed on any output channel.

The reasons for putting program output into a George file rather than merely printing it are

- 1 The file can subsequently be used as an input file to another program. Indeed, it may never be necessary to obtain results as a physically printed document or paper tape.
- 2 Once the output is in a file, it can be printed out or punched on paper tape or cards at any time by using the LISTFILE command (George 5). This command need only be given if and when the physical output is required - for example when the contents of the file are known to be correct.

To send output to a file, an ASSIGN command is used instead of LIST, as described in the next section.

4.2.5.1 Putting output into a file

To send character output to a George file, an ASSIGN command for the output channel is used between COMPILE and EXECUTE. For the standard output channel used by the "print" procedure, which is numbered zero, the command is

ASSIGN *1p0, filename

This allows up to 500 lines of output (otherwise see 4.2.5.2).

For other channels of output, a more general form of the ASSIGN command is used:

ASSIGN channel type channel number, filename (qualifier)

The channel number, equal to or greater than 1, is that associated with the charput variable in the procedure "openc". The channel type and qualifier depend on the second parameter given to openc (described in LIBRARY Input/Output 2.5) and is shown in the following table.

openc channel type	ASSIGN channel type	qualifier
file printer	*fw	GRAPHIC
wide printer (n)	*fw	GRAPHIC
file punch	*fw	NORMAL
tape punch	*tp	NORMAL

The letters fw stand for file-writer. Whenever a NORMAL qualifier is given it is assumed that the characters NUL (blank) and DEL (all holes) are not significant. If they are, replace NORMAL by ALLCHAR.

Each ASSIGN command sets up a permanent file called filename and this will receive the output from the procedures in the program which use the charput variable associated with the channel number. If a file with the same name already exists its previous contents will be destroyed. To add output to an already existing file, or if more than 500 lines of output are required, see section 4.2.5.2. Only one ASSIGN command can be given for each output channel, and ASSIGN and LIST cannot both be used for the same channel.

4.2.5.2 Adding output to a file, or producing more than 500 lines

If more than 500 lines of output are required on a channel, a LIMIT qualifier must be added to the ASSIGN command. To allow up to integer lines of output its form is LIMIT integer, placed inside brackets after the filename, for example

```
ASSIGN *fw1, ctsfile1 (GRAPHIC, LIMIT 1000)  
ASSIGN *lp0, ctsout (LIMIT 2000)
```

It is also possible to add output at the end of an already existing file. This is achieved by giving an APPEND qualifier with the command, as in

```
ASSIGN *fw1, ctsfile1 (GRAPHIC, APPEND)  
ASSIGN *lp0, ctsout (APPEND)
```

Information generated by the program will be added to that already in the file. Care should be taken that the information added is of the same sort as the original - do not append output from a "filepunch" channel to a file made up by a "fileprinter". If more than 500 lines are to be added during the program run a LIMIT qualifier will also be necessary:

```
ASSIGN *fw1, ctsfile1 (GRAPHIC, APPEND, LIMIT 1000)  
ASSIGN *lp0, ctsout (APPEND, LIMIT 2000)
```

Any qualifiers required (including the word GRAPHIC described in section 4.2.5.1) are always given as a list inside brackets and separated by commas - they may be given in any order. Note that the maximum amount of information a file can hold is approximately 10000 lines.

4.2.6 Connecting binary files to the program

For each binary channel opened in the program an ASSIGN command must be included between COMPILE and EXECUTE. The ASSIGN connects the channel number used in the "openb" or "openarray" procedure (LIBRARY, Binary Transput) with the file which is to be read or written by the program. Such a file must be created by a CREATE command (GEORGE 5.6) before any data can be put into it. To connect it to a channel for a particular program run the ASSIGN command needed depends on the use being made of the file in the run, and is as follows:

When reading the file only

ASSIGN *da channel number, filename

When writing to the file, or reading and writing

ASSIGN *da channel number, filename (OVERLAY)

When the special option "offset" is used in "openb"

ASSIGN *da channel number, filename (OFFSET)

If the binary transput procedures are being used to read magnetic tape a totally different command is needed, and this is described in the section on magnetic tape.

4.2.7 Using switches

It is sometimes desirable to be able to change the running of a program in some way without having to input extra data. For example, when a program is being run regularly it may not be necessary to (say) print out intermediate results every time. On the other hand it would be undesirable to leave out such clauses in the program entirely as they would be needed in cases of unexpected failure. The printing (or any other action) can be carried out conditionally by means of the program's "switches". These are a group of 24 distinct boolean values, any of which can be set by a command between COMPILE and EXECUTE and tested by the "switch" library procedure in the program. The switches are numbered from 0 to 23 and are initially all "off". The George command

ON integer1, integer2

will "turn on" the switches with numbers integer1, integer2 etc (ie set them to TRUE). The similar OFF command sets the specified switches to FALSE.

A typical use might be:

In the program

IF switch (2) THEN print (test result) FI

Between COMPILE and EXECUTE

ON 2 [so that switch (2) will deliver TRUE during this program run]

Note that switch 23 should not be used if the program includes the library procedure "save".

4.3 Programs which run for more than 30 minutes

If the machine breaks down in the course of a run, all intermediate results in core are lost and filestore files which are being used for output will be reset to the state they were in when last dumped (see 5.10.2). The job could simply be repeated, but if the breakdown occurred in the middle of a very long run, this would be very wasteful of computer time. To minimise the amount of computation wasted, long running programs should periodically preserve their state in files, so that in the event of a failure they may be restarted. It may be that only a small amount of information need be preserved - an array and a few variables perhaps - and that all the input is read at the beginning of the run and all the output for the filestore produced at the end. In this case it is fairly easy to design the program so that it may be restarted. The program should be written to take initial data from a file and update this initial data as it progresses, so that when restarted with the new data very little of the calculation would be repeated. Remember that if the information is to be stored in a filestore file the file should not be attached throughout the program run or the whole point would be lost; if there were to be a breakdown the file would be restored to its state at the beginning of the job. Use the library procedure "obey command" to attach the file, store the restarting information, and then release the file by a call of closeb or closec as appropriate. If a job is designed to run and restart in this way please give clear instructions to that effect on the yellow job card.

In more complicated cases the long running program will produce output or require input continuously, and the dynamic state of the program may need to be recorded. The library procedure "save" and the macro-command SAVEEXECUTE are provided to deal with this. The program must call the procedure "save" (described in 4.3.1) so that the state of the job is safely dumped into a file periodically. Two job descriptions must be provided, as described in 4.3.2, one of which sets up the program initially and runs it until it has been saved at least once, and another which is used if it is necessary to restart the program from its last save point. In both jobs, the macro-command SAVEEXECUTE must be used in place of EXECUTE, to ensure that output files to be kept are dealt with correctly. Character and binary output from the program is initially directed by SAVEEXECUTE to temporary files, which are then used to update permanent filestore files at every save point. Once the program run has been completed these permanent files can be dealt with appropriately, for example, character files may be printed out by a LISTFILE command.

Private magnetic tapes and discs which may be in use are not dealt with by "save" or SAVEEXECUTE. The program must be written so that magnetic tapes are repositioned whenever it is restarted. It is possible that no special action need be taken for a private disc, but this depends on exactly how reading and writing to the disc is being carried out. It should be borne in mind that when a program is restarted, a portion of it (from the last save point to the point at which the breakdown occurred) is repeated. If a sequence of read and write transfers to the same address has been performed after the save point, the disc may be left in a state incompatible with the restarted program.

4.3.1 The "save" library procedure

```
PROC save = (INT count, [ ]REF CHARPUT input channels) BOOL
```

This procedure is used in conjunction with SAVEEXECUTE to bring about the periodic saving of a job. It should be written in the main loop of the program and the actual saving of the job takes place after "count" calls of the procedure. To force a save every time, the procedure is called with the first parameter equal to one. The integer supplied as "count" should be such that saving occurs at approximately 30 minute intervals. If in doubt, please consult the operations manager.

The second parameter of save should be a list of those charput variables which are controlling input channels in use at the save point. (If there is more than one channel the list is enclosed in brackets.) When the program is restarted after an interruption all its input files are re-attached in the restart job. The save procedure then re-reads these files to the same reading position as at the save point, using the list of charput variables supplied. If the procedure "read" is being used to take data from a file, remember to include its charput variable "standin". If there is no input data, or if all the data is read before calling save, an empty row may be given, as shown in the following example:

```
[1:0]REF CHARPUT no channels;  
----  
save (20, no channels);  
----
```

When the program is restarted from a save point, "save" delivers the value TRUE, whereas for all other times it delivers the value FALSE. This information is needed to program the repositioning of magnetic tapes and resetting of private discs. Each time the job is saved the message :HALTED :SV is sent to the monitoring file, together with the time used by the job so far, in minutes and seconds. A typical call of the procedure might be

```
save (50, (standin, in1))
```

which saves the job once every 50 times the procedure call is obeyed.

Programs incorporating "save" should not use switch 23 (see George 4.2.7). This switch is used in a special way at each save point to enable the computer operators to stop the job if they wish.

4.3.1.1 Saving a program which has graphical output

The procedure "endgraphs" should preferably be called at the very end of the program or, if this is not possible, immediately before a save point. This reduces the possibility of endgraphs being obeyed twice as the result of a restart, which would cause any offlined graphs to be plotted twice. Any binary file used for the graphical output must have a generation number of 1 and the channel should be given as a parameter to SAVEEXECUTE, as for any other output file.

4.3.2 Job descriptions for long running programs

Two job descriptions are needed for running a program which uses "save". One is an initial job to COMPILE (or GETCOMPILED) the program and start it running, and the other is a job which can be used to restart from a save point, should this be necessary. It is usual to test the restart job before submitting an actual long run. Each job must use SAVEEXECUTE for running the program instead of the normal EXECUTE. The SAVEEXECUTE command takes a variable number of parameters, depending on how many output channels are to be dealt with. Its form is

SAVEEXECUTE filename, ((channel),filename), ((channel),filename) etc

The filename given as the first parameter must be a simple file identifier (George 1.4) and SAVEEXECUTE creates several files of this name (with differing language codes). These files are used to hold the state of the program and exist from the time the job is first put in until the program finally finishes, when they are erased by SAVEEXECUTE. The other parameters are to ensure that all output files from the program are maintained during the run. A parameter must be given for each output channel in use, including any binary channels using filestore files which are being written to. The channel in the parameter depends on the type of output and should be as follows:

For the default channel used by the procedure "print"

*1p0 [the 0 is a zero]

For a channel opened as a "fileprinter" or "wideprinter"

*fw channelnumber

For a channel opened as a "filepunch"

*fp channelnumber

For a channel opened as a "tape punch"

*tp channelnumber

For a binary channel which is being used for writing

*da channelnumber

The channelnumber in each case is the integer used in the "openc", "openb", or "openarray" procedure. All output on the channel will be put into a file of the name given in the second part of the parameter; character files (but not binary files) will be created if necessary. All the files must belong to the user running the job. Any output channels which are not given in the list must be used and closed before the first or after the last save point.

Except in the case of TIME UP, a failure in the program will cause diagnostics to be printed and the job ended in the usual way. It will not be possible to restart. For TIME UP the job is simply ended, so that it could be restarted from the last save point. Please make sure that the program is not looping before doing this.

The two job descriptions incorporating SAVEEXECUTE are written as shown in the following example, which runs a long program using two channels of input (the standard channel and one extra) and two channels of output (the standard channel used by "print" and one binary channel).

The initial job

```
JOB ctslongjob, :manuals           see note a
WHENEVER COMMAND ERROR, ENDJOB ALL
JOBTIME 65 MINS                  see note b
COMPILE sourcefile                see note c
TIME 60 MINS                     see note d
STORE 6000                        see note e
ASSIGN *tr0, datafile             see note f
ASSIGN *fr1, extradata
SAVEXECUTE savedprog, ((*lp0),outfile), ((*da10),graphfile)
LISTFILE outfile, *lp              see note g
ENDJOB
*****
DISENGAGE
```

The restart job

```
JOB ctslongjob, :manuals           see note a
WHENEVER COMMAND ERROR, ENDJOB ALL
JOBTIME 65 MINS                  see note b
RESTART savedprog                see note h
TIME 60 MINS                     see note d
ASSIGN *tr0, datafile             see note i
ASSIGN *fr1, extradata
SAVEXECUTE savedprog, ((*lp0),outfile), ((*da10),graphfile)
LISTFILE outfile, *lp              see note g
ENDJOB
*****
DISENGAGE
```

Notes

- a Please give the same jobname to the initial job and the restart job.
- b Each job must include a JOBTIME command giving enough time for the complete program run.
- c A command to retrieve the compiled program from an album could be used instead. If the program is to be overlaid please see section 4.3.3.
- d Each job must include a TIME command (between COMPILE and SAVEXECUTE, or RESTART and SAVEXECUTE) giving enough time for the complete program run.
- e If a STORE command is required, it must be given in the initial job. It must not be given in the restart job.
- f Any further auxiliary commands necessary for the program run must be given here, except for the output files dealt with by SAVEXECUTE.
- g The program run may be completed in the initial job (if it is never necessary to restart) or in the restart job. When it is finished the command after SAVEXECUTE is obeyed. At this point (in both jobs) commands can be included to (say) print out all the character output files before ending the job.
- h The special RESTART command must be used to start the program from its last save point. Its parameter is the same as the first parameter of SAVEXECUTE.
- i Commands must be given between RESTART and SAVEXECUTE to connect all the input files for the program (including any binary files which are being used solely for reading). Output files dealt with by SAVEXECUTE must not be connected.
- j For short testing runs it is possible (and more efficient) to run a program including "save" with EXECUTE rather than SAVEXECUTE. If the program attempts to save itself when run in this way a fault will occur.

4.3.3 Saving an overlaid program

An overlaid program uses an extra file to hold the sections of program which are to be overlaid. This file is normally provided and connected automatically by the system, but if the overlaid program is to be saved the file must be provided by the user, as a file on a private disc. Please ask at computer reception if you do not already have a suitable disc and file. An extra command must then be given between GETOVERLAY and SAVEEXECUTE in the initial job, and between RESTART and SAVEEXECUTE in the restart job. This extra command is

ONLINE *da13(OVERLAY), (csn,filename)

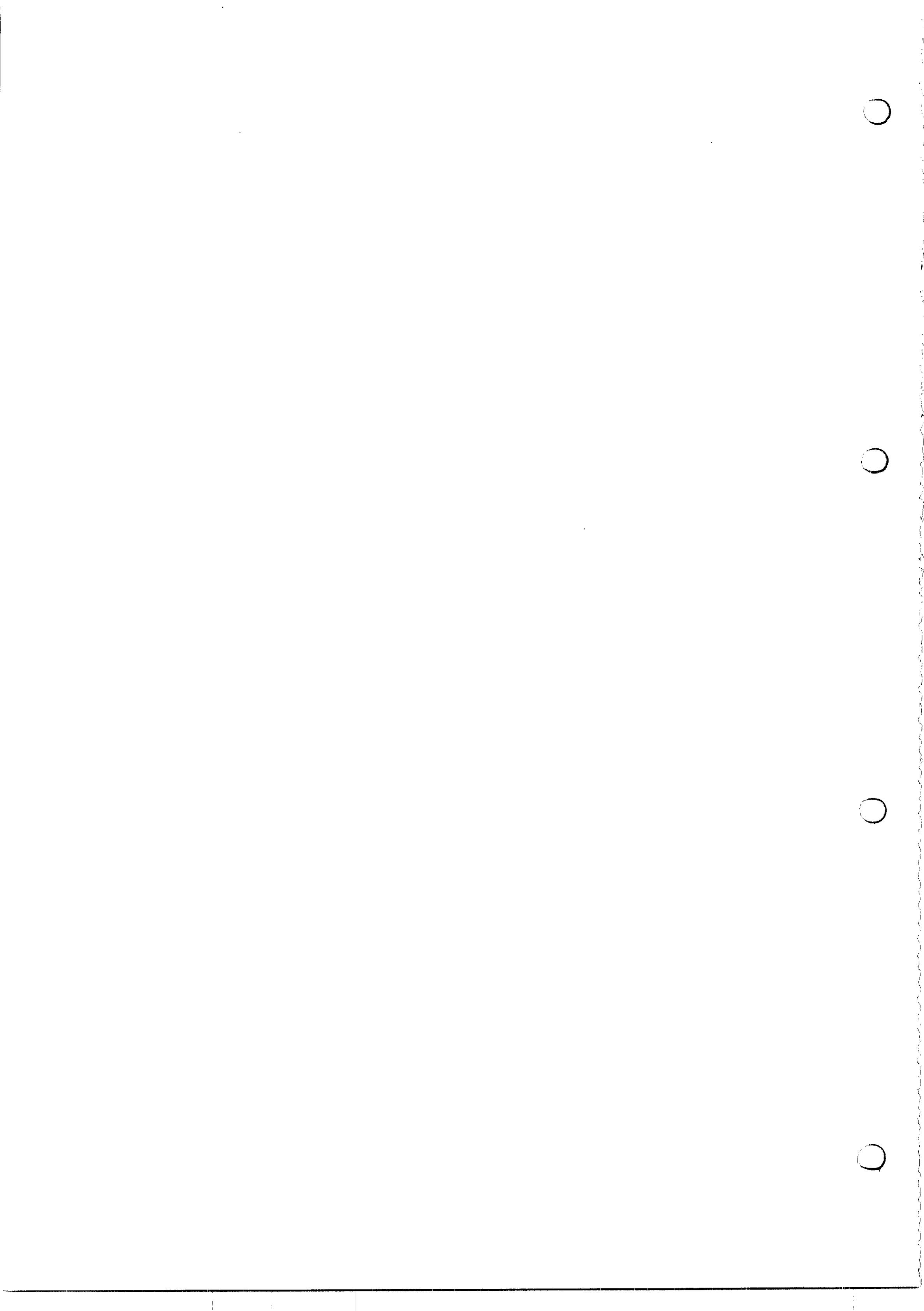
where csn is the cartridge serial number of the private disc and filename is the name of the file.

UTILITIES

Contents

ALBUMS

- 1 Using albums
 - 1.1 Using segments from an existing album
 - 1.1.1 Using segments from more than one existing album
 - 1.2 Preserving a compiled program
 - 1.2.1 Principles of initializing an album
- 2 How to segment a program
 - 2.1 Terminology - preliminary segments, final segment and program
 - 2.2 Segmentation and the use of KEEP
 - 2.3 Assembly of a complete program
 - 2.4 Example of a segmented program
 - 2.5 Album hierarchy
- 3 The album utility programs
 - 3.1 Creating and updating an album - the UPDATEALBUM command
 - 3.1.1 Updating instructions
 - 3.1.2 Example of program segmentation, with job description
 - 3.2 Extending an album
 - 3.3 Copying or renaming an album
 - 3.4 Obtaining information about an album - the ALBUMPRINT command



ALBUMS

An album is a special kind of George file used for holding programs or portions of programs in compiled form. Many users first encounter albums when they need to use special facilities which are not in the system library. For example, there is a graphical output package which has already been compiled into an album for use by all who need it. Another common use of albums is for preserving users' own programs, so that they can be run repeatedly without having to compile them afresh every time. These uses of albums are described in section 1.

To take advantage of albums to the fullest extent, it is necessary to learn how to segment a program, compile the segments separately and subsequently link them together. A user may then build up a private library of procedures, etc, for use in different programs. Details are given in section 2.

The necessary facilities for album management are provided by special album Utility Programs, details of which are given in section 3.

UTILITIES

Albums

1

1 Using albums

1.1 Using segments from an existing album

To write a program which uses procedures, mode names, etc, defined in an existing album, the program must start with a heading which lists the names of the required segments and the album containing them. The program takes the form

```
title WITH segmentname, segmentname, ... FROM albumname
BEGIN
    body of program
END
FINISH
```

The segmentnames are those of the segments in which the items to be used were declared. The documentation of the album will specify which these are, and may also specify the order in which they must be listed, if this is important. The albumname is the George filename of the album, ie :user.filename when the album belongs to another user. Only one album name can be given, but it is possible to use segments from different albums in one program by setting up a new album and initializing it to 'contain' the albums required, as described in the next section.

1.1.1 Using segments from more than one existing album

A program may require to use segments from more than one existing album. As the heading can only mention a single album after the word FROM, the user must first set up a suitable album which 'contains' the required existing albums. This is done by creating an album with the CREATE command as described in section 3.1, and then initializing the album (by means of the UPDATEALBUM utility) to the required existing albums. The WITH-list of any program can now include segments from any of the existing albums contained in the new album, with the name of the new album given after the word FROM.

The order of the segments in the WITH-list must correspond to the order in which the existing albums were given when the new album was initialized. For example, if "newalbum" was initialized to albums A and B in that order, all required segments from A must appear in the WITH-list before any from B. The ordering of the segments in any one group may or may not be important; the documentation of the individual albums will specify the order required, if a special order is necessary.

1.2 Preserving a compiled program

Any program which has just been compiled will soon disappear from the system file which holds it initially. To preserve it for repeated running, it must be moved from the system file into an album belonging to the user. Any number of different compiled programs can be preserved in one album. The actual moving of a program is done by the UPDATEALBUM command, fully described in section 3.1.

If the user does not already possess an album, he can easily acquire one by means of the CREATE command, as described in section 3.1, after which the album file must be initialized. This is done when the UPDATEALBUM utility is used on the new album file for the first time, and is an essential preliminary. The actual mechanism of initialization is described under the UPDATEALBUM utility, but the purpose of initialization should be understood clearly first, and is described in section 1.2.1 below.

Once a program has been compiled and put into an album, it can be retrieved and run at any time by the command

GETCOMPILED segmentname, albumname

followed by auxiliary commands and EXECUTE in the ordinary way, as described in George 4.1.1. The source text of the program should of course be kept as well - there is no anti-compiler to reconstitute it!

UTILITIES

Albums

1.2.1

1.2.1.1 Principles of initializing an album

In the simplest case, an album is initialized to the system library alone. This provides minimum facilities; it enables the preserved program to use any system library segments it may require. The album can be imagined to 'contain' all of the compiled segments in the system library before the user puts in his own program. In fact, a system of cross-referencing is set up by the initialization process, so the library segments are not actually copied into the user's album.

Similar principles apply when the programs to be preserved require segments from other existing albums. The user's album must be initialized to 'contain' these other albums. Suppose, for example, that a program uses various segments from the album :devlib.graphslib. The user's own album must then be initialized to :devlib.graphslib, and the heading of the program might then be

```
myprog WITH graphfile, pictures, sketch graphs,  
plotnumber, micro plotter  
FROM myalbum
```

Here, "myalbum" is the name of the user's own album which, as a result of the way it was initialized, can be said to 'contain' all of the segments from :devlib.graphslib, and (always) the system library as well.

It may have happened that "myprog" was first compiled (and successfully run) with the heading

```
myprog WITH graphfile, pictures, sketchgraphs,  
plotnumber, micro plotter  
FROM :devlib.graphslib
```

If it is only now decided to preserve this program in "myalbum", the album name should be edited to read "myalbum" and the program recompiled and put into "myalbum". The point to notice is that the album name after FROM must be the name of the destination album. This rule has not always applied, but is essential when an album is initialized to more than one existing album. It is therefore recommended that it be observed in all cases.

2 How to segment a program

2.1 Terminology - preliminary segments, final segment and program

Any program which uses previously compiled segments (other than system library segments, which are looked after automatically) will have the form

```
title WITH segmentname, segmentname, ... FROM albumname
BEGIN
    body of final segment
END
FINISH
```

The segments listed after WITH are known as 'preliminary segments', whilst title is the name of the 'final segment'. The distinction is important, for although all segments are written according to the same rules of Algol, the final segment of a complete program has a special significance. Its title serves as the name, not merely of the final segment, but of the program as a whole, and the WITH part of the heading governs the assembly of all the segments required.

UTILITIES

Albums

2.2

2.2 Segmentation and the use of KEEP

A preliminary segment may consist only of declarations or it may contain unitary clauses to be obeyed. However it is written, it will almost certainly be found that some of the declared names will be needed in subsequent segments. Such names, which can be identifiers, mode names or operators, must be listed after the word KEEP before the FINISH of the segment in which they are declared. As an example, a preliminary segment could be

```
specials
BEGIN
    MODE SIGNAL = ... ;
    REF SIGNAL none = NIL;
    PROC detect = ... ;
    OP + = ... ;
    PRORITY '*' = 7;
    OP '**' = ... ;
    INT year := 365;
    SKIP {a serial clause must always end with a unitary clause}
END
KEEP SIGNAL, none, detect, +, '**', year
FINISH
```

Kept names must always have been declared at the outermost level of the segment. In this example, it happens that every one of the declared names is kept, which will not always be necessary. Note that any values associated with the kept items will carry across to later segments, eg the operator priority setting declared above, and the value assigned to "year". This is a consequence of the fact that a segmented program is assembled into one completely unified program.

To use previously kept identifiers, mode names and operators, the name of the segment which kept them must be given after the word WITH. For example,

```
laterseg WITH specials FROM albumname
BEGIN
    SIGNAL s;
    year := 366;
    etc
END
FINISH
```

As this segment uses the mode name SIGNAL, "specials" must be included in the WITH-list to complete the connection between the segment which kept the source-text name and the segment which uses it. When "laterseg" is compiled, "specials" must already be in the album.

2.3 Assembly of a complete program

The final segment of a program controls the selection of all the required segments. Those included in the final assembly are all those listed after WITH in the final segment, plus any others in the WITH-lists of those, and so on. The order in which all the segments are assembled is always the same as the order in which they were put into the album, and it is good practice to adhere to this order in every WITH-list. No segment will be included twice over when the complete program is assembled, no matter how many times it was mentioned in the various WITH-lists. Any library segments required are embodied with the program automatically, and come first of all.

If a preliminary segment has no KEEP list, its name would never appear in subsequent WITH-lists for name-linkage purposes, but it would have to be included in the heading of some later segment, or it would be left out altogether.

2.4 Example of a segmented program

The following example illustrates a program with two preliminary segments and one final segment.

```
firstseg
BEGIN
    CHARPUT in1, in2;
    openc(in1, file reader, 1);
    openc(in2, file reader, 2)
END
KEEP in1, in2
FINISH
```

```
inputseg WITH firstseg FROM myalbum
BEGIN
    MODE DATA = ... ;
    DATA x, y;
    get(in1, x);           Note: firstseg is included in the heading
    get(in2, y)            because in1 and in2 are used here
END
KEEP DATA, x, y
FINISH
```

```
prog WITH inputseg FROM myalbum
BEGIN
    Note: the body of this final segment cannot use the names in1 or
    in2, because the heading does not include "firstseg". However, it
    can use the names DATA, x and y.
END
FINISH
```

When "prog" is compiled, all three segments will be assembled in the order shown, which (in this instance) must necessarily have been the order in which they were put into the album.

2.5 Album hierarchy

The album in which a segmented program is held can be initialized to other existing albums, as described in section 1.2.1. Any segment of the user's program can then request segments from any of these other albums. The album name after FROM must be that of the user's own album, which 'contains' the albums to which it was initialized. The WITH-list must give the required segments in an order corresponding to the initialization sequence, ie segments from the first album, then any from the second, etc, with the user's own preliminary segments last of all. This is the only order in which the whole program can be assembled and obeyed.

A user may possess more than one album, in order, for example, to keep different areas of work distinct. It is a good plan to maintain the independence of such albums, but if a program should require some segments from each, difficulties are avoided if they have been written so that the order in which they are obeyed is unimportant. If, in spite of this recommendation, it is necessary that sets of segments from two albums be obeyed in a definite order, the second set should be in an album which was initialized to the first only.

It may happen that a user possesses an album, "oldalbum", which was suitably initialized for the programs already held in it, but he now wishes to extend the number of 'contained' albums. Ideally, the album should be re-initialized, and as this destroys its entire contents, all required segments must be recompiled and put in again. If this solution is impossible, there is an alternative though less satisfactory method of proceeding. A new album ("newalbum") is created and initialized to the extra albums and to "oldalbum". The order of the various albums is now as follows: extra albums (in order of initialization), then the albums to which "oldalbum" was initialized, then "oldalbum" itself, and finally "newalbum". New segments should now be put into "newalbum".

3 The album utility programs

3.1 Creating and updating an album - the UPDATEALBUM command

To set up an album, the user must create, once and for all, a suitable George file of size integer thousand words by the command

CREATE filename (*da, BUCKET1, KWORDS integer)

As a rough guide, integer could be 10 or 20, which should be sufficient in the first instance. The size can be increased later, but not reduced. CREATE is the standard George command for a binary file, and when it is used for creating an album file, the name of the file must be a simple George identifier. Thereafter, all operations on the album are carried out using a special utility program, called into play by the George macro-command UPDATEALBUM filename, where the file name is that of the album. This command is followed by updating instructions for the utility, each on a separate line, the last line of which must be END, which returns control to George:

```
UPDATEALBUM filename
  updating instruction
  updating instruction
  ....
END
```

3.1.1 Updating instructions

CLEARTO album1, album2,

This initializes a new album so that it 'contains' album1, album2 and any further albums given, with commas between. If an album belongs to another George user, its name must be given in full (:user.filename). The albums may usually be listed in any order, but it will often be important to remember what the order was (see 1.1.1).

If the album being initialized is not a new one, the CLEARTO will destroy its contents and make a fresh start of it. If any album in the initialization list already contains other albums, it is as though these were inserted immediately before it in the list. The utility will print the complete list in response to the CLEARTO instruction.

CLEARTO :0 [colon zero]

Initializes the album to contain only the system library.

PUTIN segment

where segment is the name of a recently compiled segment not already in the album. Puts it in. If the segment has a heading, the album name after FROM should be the name of the album being updated (see 1.2.1).

FORGET segment

Necessary before a later version of segment can be PUTIN (but see note below), or if segment is no longer required. Deletes it.

Note: FORGET is usually applied to the final or only segment of a program. When a preliminary segment is to be changed, AMEND should be tried, as deleting the segment and putting it in again will upset the order of segments in the album.

AMEND segment

where segment is the name of a recently compiled segment having the same name as a segment already in the album. If possible, the old segment will be replaced by the new one without affecting the order of segments in the album. If, however, particular changes have been made to the segment, such as alteration of WITH or KEEP lists or use of different library segments, the utility program will not allow the amendment. A message will be displayed to this effect, and the old version will have been left undisturbed. This means that the change proposed would invalidate any already compiled segments using the one to be changed.

If a segment cannot be amended, it can only be changed by a FORGET instruction followed by a PUTIN. The newly put in version will now come after all other segments in the album, and any segments which had depended on the old version cannot be incorporated in a program with the new version without recompilation. Any attempt to do so will give the message OUT OF DATE SEGMENT USED. The following steps must now be carried out:

- 1 FORGET all segments which used the old segment
- 2 Recompile these and PUTIN again. The recompilation must, of course, take place after the new version of the changed segment is in the album.
- 3 Repeat steps 1 and 2 for all segments which used those, etc.

(Consistency checks are performed at the end of each compilation, and may produce the message ALBUM CORRUPT if a mistake has been made in this sequence.)

A history of all segments in an album and information about which other segments they use can be obtained by using the ALBUMPRINT utility described in section 3.4. If other users have initialized their albums to the one being altered, it is particularly important to consider changes carefully. Recompilation should be minimized by the use of AMEND whenever possible, and if recompilation is done, any others concerned should be notified.

END

This must always be the last updating instruction. Deletes the updating program and returns control to George.

UTILITIES

Albums

3.1.2

3.1.2. Example of program segmentation, with job description

```
INPUT :manuals, sgbseg1
```

```
SEG1
```

```
'BEGIN' 'INT' P := 7;  
      PRINT((P, NEWLINE))
```

```
'END'
```

```
'KEEP' P
```

```
'FINISH'
```

```
****
```

```
INPUT :manuals, sgbseg2
```

```
SEG2 'WITH' SEG1 'FROM' SGBALBUM
```

```
'BEGIN' 'INT' Q := P + 1;  
      PRINT((Q, NEWLINE))
```

```
'END'
```

```
'KEEP' Q
```

```
'FINISH'
```

```
****
```

```
INPUT :manuals, sgbprog
```

```
SGBPROG 'WITH' SEG1, SEG2 'FROM' SGBALBUM
```

```
'BEGIN' 'INT' R := P + Q;  
      PRINT((R, NEWLINE))
```

```
'END'
```

```
'FINISH'
```

```
****
```

```
JOB sgbtest, :manuals
```

```
CREATE sgbalbum(*da, BUCKET1, KWORDS 20)
```

```
COMPILE sgbseg1
```

```
UPDATEALBUM sgbalbum
```

```
    CLEARTO :0
```

```
    PUTIN seg1
```

```
    END
```

```
COMPILE sgbseg2
```

```
UPDATEALBUM sgbalbum
```

```
    PUTIN seg2
```

```
    END
```

```
COMPILE sgbprog
```

```
UPDATEALBUM sgbalbum
```

```
    PUTIN sgbprog
```

```
    END
```

```
GETCOMPILED sgbprog, sgbalbum
```

```
CHECK
```

```
EXECUTE
```

```
ENDJOB
```

```
****
```

```
DG
```

The following results were output:

```
+7
```

```
+8
```

```
+15
```

Continuation at a MOP terminal:

```

login sgb-ro8, :manuals
edit sgbseg1
ts/pr/, a/((/"p=",/, e
compile sgbseg1
updatealbum sgbalbum
UPDATE ALBUM :MANUALS.SGBALBUM
    amend seg1
AMENDED SEGMENT NOW REQUIRES LIBRARY SEGMENT LETTERSOUT
DISPLAY : AMENDED SEGMENT HAS DIFFERENT SPEC
:HALTED : FL
DISPLAY: TRY AGAIN OR TYPE "END" (WITHOUT QUOTES)
    forget seg1
FORGOTTEN SEGMENT SEG1
    putin seg1
NEW SEGMENT SEG1
    end
END
    getcompiled sgbprog, sgbalbum
    check
    execute
:HALTED :L2
DISPLAY: OUT OF DATE SEGMENTS USED
DELETED
    compile sgbseg2
    updatealbum sgbalbum
    forget seg2
    putin seg2
    forget sgbprog
    end
    compile sgbprog
    updatealbum sgbalbum
    putin sgbprog
    end
    getcompiled sgbprog, sgbalbum
    check
    execute
P=      +7
      +8
      +15
DELETED
    edit sgbseg1
    t1, r/7/6/, e
    compile sgbseg1
    updatealbum sgbalbum
    amend seg1
AMENDED SEGMENT SEG1
    end
    getcompiled sgbprog, sgbalbum
    check
    execute
P=      +6
      +7
      +13
DELETED
    logout

```

Commands and updating instructions are shown indented and in lower case. A selection of the system responses is shown in upper case. These responses may vary in detail from time to time.

UTILITIES

Albums

3.2

3.2 Extending an album

An album can be increased in size, at any time after it has been initialized, by the sequence

```
UPDATEALBUM filename
INCREASE KWORDS integer
further updating instructions if required
END
```

This increases the size of the album file filename by (integer * 1024) words. The contents of the album are not disturbed.

3.3 Copying or renaming an album

Part of the information in an album is its own filename in full (ie :user.filename) and the file names of any albums to which it is initialized. If an album file is renamed or copied to another file (by a George command RENAME or COPY) the name inside the album must be altered to match, and this can be done by a special use of UPDATEALBUM. This step is necessary before any segment using the new or renamed album can be compiled. The updating sequence required is

```
UPDATEALBUM newfilename
RENAME
END
```

If any other albums have been initialised to the renamed one (by CLEARTO ... oldname ...), these albums in turn must be updated by the use of UPDATEALBUM with the instruction

RENAME WITH ... ,newfilename, ...

The list of album names after RENAME WITH is the list originally given after CLEARTO, but with the old album name replaced by the new. This process must be repeated for any album initialized to this second one, and so on, even though the originally renamed album may not appear explicitly in the list. The altered file names are printed out each time, and should be checked.

5.4 Obtaining information about an album - the ALBUMPRINT command

To obtain data about the current state of an album, a separate utility program must be called by the command

ALBUMPRINT filename

followed by steering instructions from the following list, each on a separate line. The last line of all must be END, which returns control to George.

ALL	Prints size of album file, names of any other albums to which the album is initialized, and names of segments in an approximate alphabetical order which has no special significance.
HISTORY	Prints names of all segments in the album, including those now forgotten (indicated by the word FORGOTTEN), in <u>reverse</u> order to that in which they were filed.
CROSSREF	Prints the names of other segments (including library segments) used by each segment in the album. Also gives the names of any other albums to which the album is initialized.
SPACE	Prints space occupied by each segment (in reverse order to that in which segments are filed), and total available space in album.
PRINTSPEC <u>segment</u>	Prints name of <u>segment</u> , names of other segments it uses (including library segments), and names and modes of all kept items.
END	Deletes the utility and returns control to George.

The information asked for is produced on a separate line printer document.

Q

O

O

Q

LIBRARY

Unless otherwise stated, all modes, operators and identifiers described in this section are available automatically, without declaration, to any Algol 68-R program.

CONTENTS

APPROXIMATION

- 2 Standard REALPAIR constant for tests of approximation
Tests of approximation

BINARY TRANSPUT

- 1 Introduction
2 Using backing store
 2.1 Binary channels and TRANSIT variables
3 The channel opening procedure "openarray"
4 The channel opening procedure "openb"
 4.1 Buffered transput
 4.2 Direct transput
5 Transput procedures and addressing
 5.1 Random addressing
 5.1.1 Simultaneous address setting and transput
 5.2 Serial addressing
 5.3 Finding out the current address
6 Groups
 6.1 Skipping over groups in a serial file
7 Direction of transput; input, output or both?
 7.1 Common buffering for input and output
8 Working with several transit variables
9 Procedures using a previously selected transit variable
10 Saving program space
11 Channel closing
12 Special disc procedures
 12.1 Finding out a file name
 12.2 Using a non-standard bucket size
 12.3 Altering the size of a file
 12.4 Renaming a file
13 Faults and event procedures
 13.1 The standard event settings and how to change them
 13.2 Event numbers and associated error messages
 13.3 Notes on particular events
 13.4 Transput within an event procedure and warning on scopes
14 Magnetic tape
 14.1 Opening a magnetic tape channel
 14.1.1 Grouping of data on magnetic tape
 14.1.2 Direction of transput on magnetic tape
 14.2 Accessing magnetic tape
 14.2.1 Tape marks
 14.2.1.1 The tape mark event
 14.2.2 Addresses on magnetic tape
 14.2.3 Special tape positioning procedures
 14.3 Writing after reading
 14.4 Tape names
 14.5 The end of the tape
15 Summary of options for openb

LIBRARY

Contents

2

CONSTANTS

DIFFERENTIAL EQUATIONS

- 1 Integration of ordinary differential equations (Runge-Kutta-Merson)

FITTING AND LEAST SQUARES ANALYSIS

- 6 Fitting (x,y) points with given order polynomial
- 7 Fitting (x,y) points with polynomial of adaptable order
- 8 Linear least squares fitting
- 9 Solution of non-linear simultaneous equations with computable derivatives
- 10 Solution of non-linear simultaneous equations whose derivatives cannot be calculated
- 11 Minimization of a sum of squares of functions whose derivatives cannot be calculated

FUNCTIONS

- 1 Elementary functions
Complex functions
- 2 Bessel functions
- 3 Error function
Complementary error function
- 4 Gamma function
- 5 Evaluation of real polynomial
- 6 Complete elliptic integrals
Ratio of complete elliptic integrals of the first kind, and its inverse
- 7 Conversion of binary to Gray code and vice versa
- 8 Highest common factor and least common multiple of two integers
- 9 Complex error function
Complex complementary error function
Complex Φ -function
- 10 Exponentiation of a complex number
- 11 Fresnel integrals
- 12 Jacobian elliptic functions
- 13 Sum of Chebyshev series

GRAPHICAL OUTPUT

- 1 Introduction
 - 1.1 An example program
 - 1.2 Obtaining graphical output
- 2 Some modes and operators
- 3 Primitive sketches
 - 3.1 Point plotting symbols
- 4 Trails
 - 4.1 Drawing curves
 - 4.1.1 Inaccuracies in curve drawing
 - 4.2 Symbols and other extensions to trails
- 5 Drawing graphs
 - 5.1 A graph of values known at equally spaced argument values
 - 5.2 A graph given as a set of positions in an array
 - 5.3 Drawing an analytical smooth curve
- 6 Transformations
 - 6.1 Compound transformations
 - 6.2 Logarithmic scales
- 7 Different types of line drawing
 - 7.1 Broken lines
 - 7.2 Other line types
- 8 Drawing pictures
 - 8.1 Sketch fitting
 - 8.2 Picture headings and margin axes

GRAPHICAL OUTPUT (contd)

- 9 Text in graphical output
 - 9.1 Text as a sketch
 - 9.2 Numbers and the use of CHARPUT variables
 - 9.3 Other character fonts
 - 9.4 Picture annotation
- 10 Drawing axes
- 11 Starting and finishing
 - 11.1 Picture information on backing store
 - 11.2 Picture information in program data space
 - 11.3 Finishing
 - 11.4 Picture space
 - 11.5 Using pre-defined sketches
- 12 Graphical output devices
 - 12.1 Off-lined plotters
 - 12.2 Line printer pictures
 - 12.2.1 Line printer pictures off-lined
 - 12.3 The pen plotter
 - * 12.4 The microfilm plotter
 - * 12.5 Contraves Coragraph
 - 12.5.1 Obtaining output for the Coragraph

INPUT/OUTPUT

- 1 Simple Input/Output
 - 1.1 The procedures read and print
 - 1.2 Standard forms for input items
 - 1.2.1 Input of numbers
 - 1.2.2 Input of non-numerical items
 - 1.2.3 Input of arrays
 - 1.3 Standard form of printing
 - 1.3.1 Items printed with automatic layout safeguards
 - 1.3.2 Items with no layout safeguards
 - 1.3.3 Printing of arrays
 - 1.4 Spaces, new lines and new pages
 - 1.4.1 Space and backspace
 - 1.4.2 Newline and newpage
 - 1.5 Changing printing styles
 - 1.5.1 Numberstyle
 - 1.5.2 Reading numbers containing gaps
 - 1.5.3 Further settings of "sign OF numberstyle"
 - 1.5.3.1 Printing of leading zeros and spaces
 - 1.5.3.2 Printing to non-decimal bases
 - 1.5.3.3 Reading integers using non-decimal bases
 - 1.5.4 NUMBERSTYLE variables
 - 1.6 Fault displays

* Issued to RRE users only

sgb(3.76)x

INPUT/OUTPUT (contd)

- 2 Charput Variables
 - 2.1 Procedures "get" and "put"
 - 2.2 Extra channels of input
 - 2.3 Extra channels of output
 - 2.4 Opening and closing channels
 - 2.4.1 An example of multi-channel input/output
 - 2.5 Use of special channel types
 - 2.5.1 Long lines of printed output
 - 2.5.2 Data in long lines (few line feed characters)
 - 2.5.3 Reading and writing non-graphic-set characters
 - 2.5.4 Output on paper tape
 - 2.5.5 Input from and output to []CHAR variables
- 3 Layout Facilities Using Charput Variables
 - 3.1 Library layout procedures
 - 3.2 Positional information
 - 3.3 Constructing layout procedures for use in read and print
 - 3.4 The procedure "standput" and re-setting of numberstyle
 - 3.5 Reading rows of characters - arbitrary terminators
- 4 Formats
 - 4.1 The procedures "inf" and "outf"
 - 4.2 Standard number patterns
 - 4.3 Insertions and replication
 - 4.3.1 Examples of insertions
 - 4.4 Replication of pictures
 - 4.5 Dynamic replication
 - 4.6 Construction of number patterns from frames
 - 4.6.1 Placing of the sign
 - 4.6.2 Definitions of the standard number patterns
 - 4.6.3 Left justification on output
 - 4.6.4 Frame suppression and omission of leading zeros
 - 4.7 Special types of number
 - 4.7.1 Long integers
 - 4.7.2 Radix other than 10
 - 4.8 Ignoring spaces and new lines
 - 4.9 Booleans and characters
 - 4.10 Choice patterns
 - 4.10.1 Boolean choice pattern
 - 4.10.2 Integer choice pattern
 - 4.11 The procedures in, out and format
 - 4.11.1 Clear format
 - 4.11.2 Advancing the position in a format
 - 4.12 Identification of formats
 - 4.12.1 Formats within formats
 - 4.12.2 Multi-channel transput with formats
 - 4.13 Index of formatting symbols
 - 4.14 Glossary of formatting terms

- 5 Events
 - 5.1 The event field of a charput variable
 - 5.2 Changing the event field
 - 5.3 Examples of event procedures
 - 5.3.1 Ending input when the end of a file is reached
 - 5.3.2 Continuing after a number format error on output
 - 5.3.3 Ending input of an indefinite sequence of numbers
 - 5.3.4 Reading rows of characters
 - 5.3.5 The new line and new page events
 - 5.3.5.1 The initial new page event; use of "line number" and "page number"
 - 5.4 Table of event numbers
 - 5.4.1 Non-negative event numbers
 - 5.4.2 Fixed and variable length input of characters
 - 5.4.3 Skipping an item
 - 5.5 Writing general event procedures
 - 5.5.1 Warning about scopes
 - 5.5.2 Input or output within an event procedure
- 6 Primitive input-output procedures and handling non-standard character sets
 - 6.1 Calling the procedure "in OF"
 - 6.2 Changing the "in OF" procedure
 - 6.3 Calling the procedure "out OF"
 - 6.4 Changing the "out OF" procedure
 - 6.5 Table of characters and interface codes
- 7 Advanced techniques
 - 7.1 Assigning charput variables
 - 7.2 Repositioning and reading or writing within any line of input or output
 - 7.3 Two level transput

INTERPOLATION AND DIFFERENTIATION

- 2 Interpolation in a function known at equal intervals of its argument
Interpolation and differentiation in a function known at equal intervals of its argument
- 3 Interpolation in a function known at unequal intervals of its argument
Interpolation and differentiation in a function known at unequal intervals of its argument
- 4 Solution of $f(x) = 0$
- 5 Real roots of a polynomial equation

MATRICES AND VECTORS

- 2 Solution of simultaneous equations
Solution of further simultaneous equations with same coefficients
- 3 Matrix multiplication
- 4 Diagonalisation of a real symmetric matrix
- 5 Diagonalisation of a Hermitian matrix
- 6 Scalar product of two vectors
Hermitian product of two complex vectors
- 7 Adding a fraction of a row of reals to another row
- 8 Transposing a real matrix

MISCELLANEOUS

- 1 Dates
- 2 Day and month in character form
- 3 Time of day
- 4 Terminating a program at any point
- 5 Obeying a George command from a program
- 6 Testing switches
- 7 Outputting a message to the monitoring file
- 8 Forcing a program failure
- 9 Extra action at program failures
- 9 Continuing after a program failure

MODES

OPERATORS

- 2 Arithmetical
- Arithmetical comparisons
- 3 Arithmetical assignments
- 4 Other numerical operators
- 5 Operations on booleans
- 6 Operations on arrays
- 7 Bit patterns
- Correspondence between bits and integers
- 8 Logical operations on bits
- Logical and circular shifting
- 9 Comparing bits values
- Operations on particular binary digits
- 10 Characters
- Comparing characters and sets of characters
- 11 Special bytes operators

PRIORITIES OF OPERATOR SYMBOLS

QUADRATURE

- 1 Definite integral of a function
- 2 Multiple quadrature over hyper-rectangular region
- Multiple complex quadrature over hyper-rectangular region

RANDOM NUMBER GENERATION

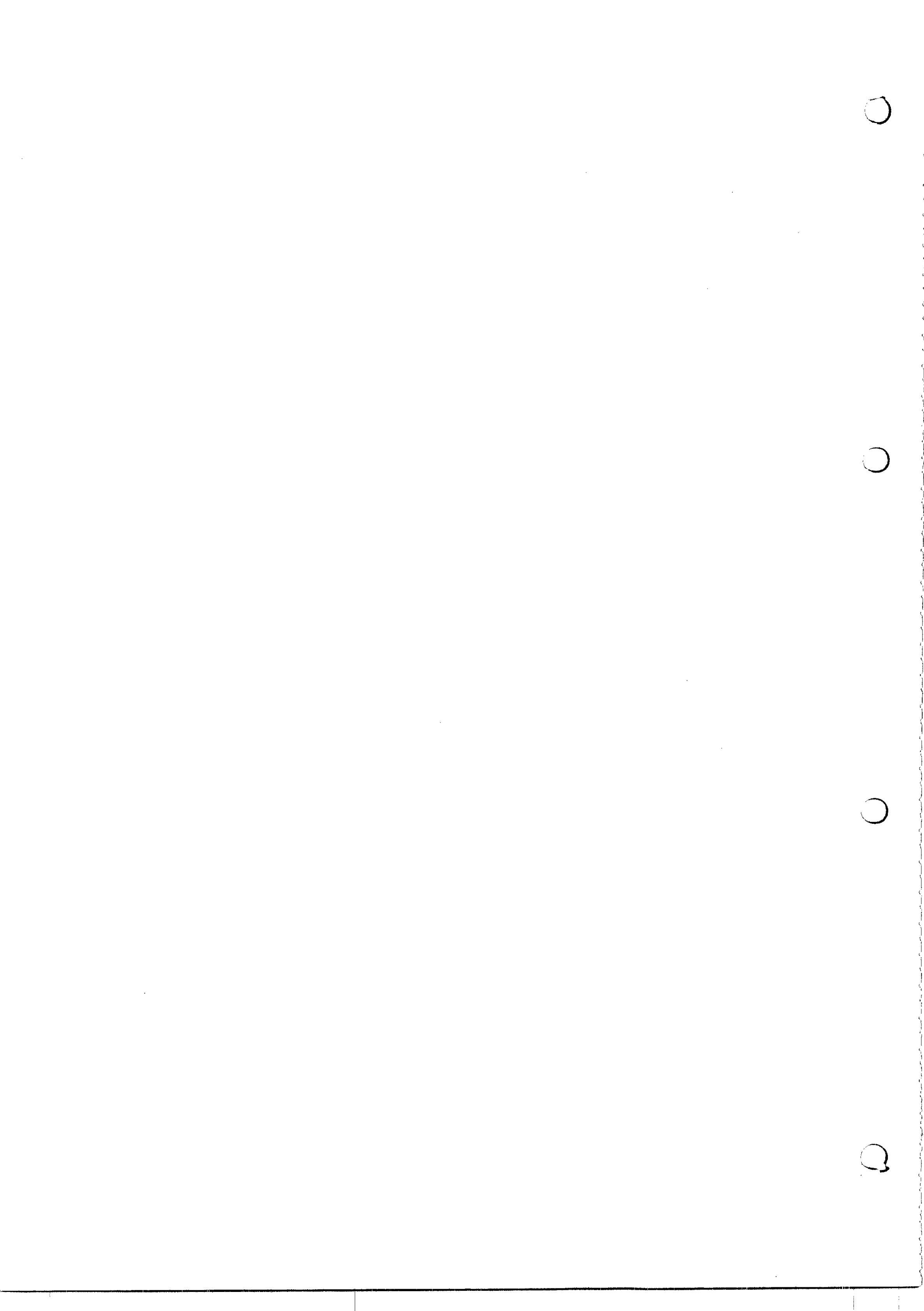
- 2 Random numbers uniformly distributed in (0.0, 1.0)
- 3 Random integers uniformly distributed
- 4 Random numbers normally distributed
- 5 Random numbers in negative exponential distribution
- 6 Random numbers in Rayleigh distribution
- 7 Random integers in Poisson distribution
- 8 Chi-squared probability distribution function
- 9 Snedecor's F probability distribution function
- 10 Fisher's Z probability distribution function
- 11 Student's T probability distribution function
- 12 Significance of correlation coefficients

SORTING

- 1 Sorting real numbers in core
- 2 Ordering arbitrary records in core

TRANSFORMS

- 1 Notes on the use of fast Fourier transforms
- 2 Analysis of complex periodic waveforms
- 3 Real periodic waveforms
- Noise analysis
- 4 Fast Fourier transform (positive imaginary exponent)
 Fast Fourier transform (negative imaginary exponent)



APPROXIMATION

Most library procedures are designed so that numerical results are given to the full accuracy of real numbers in the computer (about 11 significant figures), but procedures which work by successive approximations may have a parameter enabling the user to specify the accuracy required. By limiting the accuracy demanded in a result, computing time can be saved.

Accuracy is defined in terms of error, which can be quoted in two ways:

- (i) fractional error, e.g. 1 part in 10^4 of the result
- (ii) absolute error, e.g. not exceeding 0.000001

In practice, it is often appropriate to quote figures for both, and demand that a procedure shall produce a result in error by less than one or the other. For example, it may be reasonable to demand for the most part a fractional error of 1&-4 (i.e. about 4 significant figures), but to say at the same time that figures beyond the sixth place of decimals are of no interest, i.e. absolute errors up to about 1&-6 are permitted.

Certain library procedures have a REALPAIR parameter enabling the user to supply appropriate figures in a particular call. For example, a call of "autoquad" could be

```
autoquad(param1, param2, param3, (1&-4, 1&-6))
```

where the fourth actual parameter is a REALPAIR whose x-field is the fractional error and y-field the absolute error. The procedure autoquad will then attempt to satisfy the least stringent of the two error criteria. The library REALPAIR, "standaccuracy", can be supplied, e.g.

```
autoquad(param1, param2, param3, standaccuracy)
```

when the maximum possible accuracy is needed.

When the method of successive approximation is used outside library procedures, the programmer must provide his own criterion for terminating the repetition. A convenient way of doing this is to use the library procedure "converged". The results of the last two approximations are given to this procedure, which also takes a REALPAIR accuracy parameter of the type described above. The procedure compares the two approximations and its BOOL result may be used to decide whether or not a further round of approximation is necessary. This method works on the assumption that the best approximation so far obtained is no further from the true result than it is from the previous approximation, an assumption which is generally true for convergent processes.

LIBRARY
Approximation
2

STANDARD REALPAIR CONSTANT FOR TESTS OF APPROXIMATION

REALPAIR standaccuracy = (5&-11, 0.0)

For use as an actual parameter of any library procedure, eg autoquad, which has a REALPAIR parameter to control the accuracy of its result. The figures 5&-11 for fractional error and 0.0 for absolute error are the smallest error figures which can reasonably be supplied to a procedure, and should ensure a result accurate to about 10 significant figures.

TESTS OF APPROXIMATION

PROC converged = (REAL a,b, REALPAIR accuracy) BOOL

PROC convergedcomp = (COMPLEX a,b, REALPAIR accuracy) BOOL

These procedures compare two reals or two complex numbers respectively, to determine whether they differ, either fractionally or absolutely, by less than given amounts. The procedure delivers TRUE if either of the following inequalities is true:

ABS(a-b) < ABS(a) * fractional error
ABS(a-b) < absolute error

where fractional and absolute errors are given as the x and y fields of the REALPAIR parameter. When a and b do not agree within either of these figures, the procedure delivers FALSE. As an example,

```
IF converged(new, old, standaccuracy)
THEN GOTO end
FI
```

shows how converged may be called, when the highest reasonable accuracy is required.

1 Introduction

The term transput denotes the act of transferring data into or out of the main store of a computer. A distinction is made between character transput, which is described under Input/Output, and binary transput which is concerned with the transfer of data into or out of store without any change of representation (eg numbers remain in the scale of 2). There are two situations for which binary transput is useful:

- 1 as a way of communicating with a large auxiliary store during the running of a program.
- 2 as a way of setting up a permanent record of data which can be used by a program on several distinct runs or be used by several different programs as a common data base.

In both cases the emphasis is on program to program communication. At no time is the data converted into a form suitable for listing on a line printer.

2 Using backing store

Before any binary transput can be carried out, an area of backing store must be reserved to hold the binary data. Such an area is known as a binary file, and may be created, once and for all, by the George command

`CREATE filename (*da, BUCKET1, KWORDS integer)`

where integer is the size of file required in units of 1024 words. The size needed obviously depends on the amount of data to be put into the file, for example each integer takes 1 word, whilst each real number takes 2 words. The maximum possible is 245 KWORDS.

The actual transput of data is effected in the program by two procedures "put bin" and "get bin". The procedure "put bin" takes an arbitrary list of data objects from store and transfers them to a file; correspondingly the procedure "get bin" obtains data from a file and transfers it to a list of store variables. A single binary file can hold any mixture of data objects.

As well as the objects to be transferred, put bin and get bin must be given a control variable called a TRANSIT variable as a parameter. The TRANSIT variable (analogous to a CHARPUT variable for ordinary input/output) specifies the binary channel - and thus the file - to be used. The connection between TRANSIT variables, channels and files is described in section 2.1.

In the Algol 68-R system a binary file is divided up into 'lines' whose standard length is 128 words or 512 characters (a character is a quarter of a word). The hardware carries out a transfer in units of complete lines, starting at any specified line of the file. The procedures put bin and get bin transfer data starting at line 1 of the file and moving through the file sequentially as items are transferred. However the current positions for putting and getting can be set to any point as described in section 5.1. In other words the file can be randomly accessed.

2.1 Binary channels and TRANSIT variables

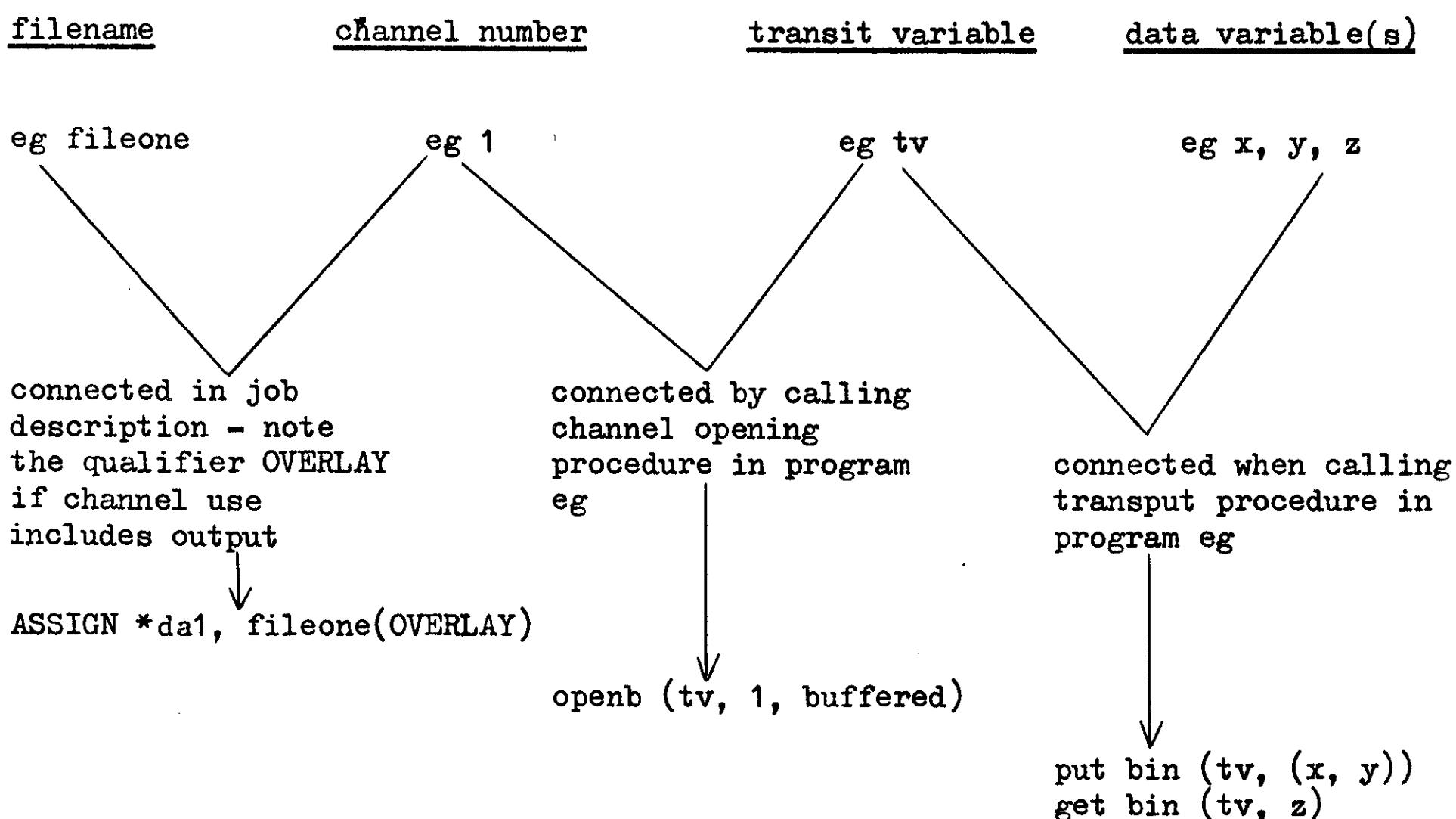
A TRANSIT variable is a structure variable used by the binary transput procedures for holding information, and one must be declared in the program for each binary channel required, for example

```
TRANSIT tv;
```

Each TRANSIT variable must then be initialised and associated with a channel number by a channel opening procedure. The channel number, in the range 0 - 10 (other channels are reserved for use by the Algol 68-R system), is used in a George command in the job description to connect the channel to the required binary file (see diagram below).

There are two alternative channel opening procedures. If a channel is to be used for storing and retrieving data in arrays only, "openarray" can be called, as described in section 3. For the transput of other data objects, transput to magnetic tape, or if arrays are to be transferred in a non-standard way, the more general procedure "openb" described in section 4 must be called. The procedure "openb" takes as a parameter a list of options which govern the way the channel is to be used; a call of openarray is equivalent to a call of openb with certain fixed options.

LINKAGE BETWEEN FILES, CHANNEL NUMBERS AND TRANSIT VARIABLES



3 The channel opening procedure "openarray"

For many scientific applications, the only binary transput facility needed is a way of storing and retrieving arrays of data. To set up a channel for this purpose, the simple channel opening procedure "openarray" can be used. The procedure takes two parameters, a transit variable and a channel number. If a transit variable, "tv" say, has been declared, the call

```
openarray (tv, 1)
```

initialises tv for array transput on binary channel 1. Thereafter put bin and get bin can be used with tv as a parameter to store or retrieve arrays of data via that channel. The elements of the array can have any of the basic modes CHAR, INT, LONG INT, BITS, REAL, BOOL, BYTES, LONG BYTES, LONG LONG BYTES or be structures composed from these modes. Thus, given an array [1:50, 1:10] REAL arr,

```
put bin (tv, arr)
```

will transfer arr (1000 words) directly to backing store, taking up as many lines (of 128 words) as necessary. An extra word is stored at the head of the array giving the total number of words transferred. If this is not equal to an exact number of lines the space at the end of the last line will be wasted (it will be filled with rubbish). The next array output will start at the beginning of the next line.

A call of get bin will retrieve an array from backing store, starting at the beginning of a line. Thus, provided that get bin is at the right place

```
get bin (tv, arr)
```

will recover data into the array arr. A check is first made that the amount of data originally transferred to the line is greater than or equal to the size of the array which is to receive it; if it is less, the fault BT-RETRIEVED GROUP IS WRONG SIZE occurs. Like put bin, get bin moves through the file from line to line as data is recovered; both start at line 1 when the channel is first opened.

Each call of put bin or get bin can transfer only one array. This can be part of a larger array, provided that the part taken occupies a contiguous area of store. The elements of an array are kept in lexicographic order of their indices (eg [1,1], [1,2], ... [1,50], [2,1], [2,2], ...), so that the slice arr[10,] is a contiguous part of the array arr and could be transferred to backing store, whereas arr[,10] could not.

The transit variable should usually be declared at the outermost level of the program. If it is not, but is declared inside a procedure, or in a nested serial clause containing a declaration, the last transput operation using the variable should be followed by the call

```
endfile (tv)
```

Enough information has now been given for the user to carry out simple array transput, as shown in the following examples:

LIBRARY
Binary Transput
3 contd

Example 1

A program to read data from a character file called rawdata and store it in a binary file called vectordata, created by the command

```
CREATE vectordata (*da, BUCKET1, KWORDS 20)

ajftransfer

BEGIN  CHARPUT char in;  openc (char in, file reader, 1);
        TRANSIT bin out;  openarray (bin out, 1);
        [1:12, 1:12]REAL a;
        TO 50 DO BEGIN
                get (char in, a);
                put bin (bin out, a)
        END
END  FINISH
```

The files would be connected in the job description by the commands

```
ASSIGN *fr1, rawdata
ASSIGN *dal, vectordata (OVERLAY)
```

Example 2

A program to read matrices from the file vectordata and calculate their eigenvalues and eigenvectors using a library procedure. The eigenvectors are written back onto vectordata, overwriting the original matrices, whilst the eigenvalues are stored on a separate file called valuedata, created by the command CREATE valuedata (*da, BUCKET1, KWORDS 10).

```
ajfcalculate

BEGIN

TRANSIT tvector, tvalue;
openarray(tvector, 1); openarray(tvalue, 2);
[1:12, 1:12]REAL a; [1:12]REAL ev;
TO 50 DO BEGIN
        get bin (tvector, a);
        symmetric qr (a, ev, 30);
        put bin (tvector, a);
        put bin (tvalue, ev)
END

END  FINISH
```

The files would be connected in the job description by the commands

```
ASSIGN *dal, vectordata (OVERLAY)
ASSIGN *da2, valuedata (OVERLAY)
```

4 The channel opening procedure "openb"

The procedure "openb" is a general purpose channel opening procedure. It has three parameters, the transit variable which is to be initialised, the number of the channel which it is to control, and thirdly a list of options showing how the channel is to be used. The options are library identifiers which determine such things as the form of data on backing store, buffering arrangements for transfers, and direction of transput on the channel. They are divided into sets, each set concerned with one aspect of use. For the buffering arrangements there are two options, buffered and direct, and one of these must always be given in the list for openb. They are described in sections 4.1 and 4.2. No explicit choice need be made from the other sets; if no option from a particular set is given, openb will automatically take the standard option in that set. These standard options assume that

transput is to a binary filestore file (or file on a private disc). For transput to magnetic tape see section 14.

both input and output may occur on the channel, since a single binary channel can handle both directions of transput. Transput may be restricted to one direction only by the use of the options described in section 7.

data will be transferred in 'groups' with a check that the amount of data to be recovered is consistent with the amount stored. This is explained in section 6.

data can be stored or recovered randomly throughout the file. This is discussed in section 5.

The options required are supplied to openb as a bracketed list, and may be given in any order. For example,

openb (tv, 2, (buffered, ungrouped))

If only direct or buffered is quoted the brackets can be omitted, as in

openb (tv, 1, direct)

The TRANSIT declaration and call of openb should normally be in the same serial clause (this is essential for buffered transput; see 4.1). In addition, if the declaration of the TRANSIT variable is not at the outermost level of the program, the procedure "endfile" must be called after the last transput using that variable, as in

endfile (tv)

Endfile is otherwise only used with buffered output (see 4.1).

4.1 Buffered transput

Buffered transput is necessary when the items of data to be transferred are small, for example single integers, reals or structures. The computer hardware and operating system take approximately 100 milliseconds to service a request for a transfer. It would thus be wasteful to transfer small data items individually. The standard way round the difficulty is to 'buffer' the data, that is to hold it in an intermediate area of store until enough objects for one line of the file have been collected, and then transfer the complete line. This is arranged by giving the option "buffered" in the list for openb. Arrays are better transferred directly, as described in section 4.2; the break-even point between direct and buffered transput comes when an item takes up at least one line.

When the buffered option is used, openb creates space for a buffer when it is called, and associates it with the transit variable. The scope of the buffer space must be the same as that of the transit variable, so the TRANSIT declaration and the call of openb must always be in the same serial clause. Failure to observe this rule could result in data overwriting the program, with undefined but disastrous consequences.

Transfers between the buffer and backing store take place completely automatically. Calls of "put bin" store data in the buffer sequentially, each item taking up as many characters (quarter-words) as necessary. Whenever a line boundary in the file is crossed a new buffer is read in, and on output the current buffer is transferred to the file. At the end of a program the current contents of any output buffer must always be transferred. This is done by calling the procedure

```
PROC endfile = (REF TRANSIT t)
```

after the last call of put bin for each transit variable. A channel need not be formally closed unless special facilities are being used (see section 11).

4.2 Direct transput

Direct transput can only be used for transferring arrays of data. When the "direct" option is given to openb each call of put bin or get bin performs a hardware transfer directly between main store and backing store. The data transferred must therefore occupy a contiguous area of store, so put bin and get bin are limited to handling one array per call.

Direct transput is the method used when a channel has been opened using "openarray", and further details can be found in section 3. A call of openb with direct as an option will only be needed if the standard options used by openarray are to be changed, for example

```
openb (tvu, 1, (direct, ungrouped))
```

arranges for arrays to be transferred without an extra word at the head. (The 'grouping' of data is described in section 6). A call of openarray is formally equivalent to a call of openb with the options (direct, wordheader).

5 Transput procedures and addressing

The basic procedures for transferring data to and from a binary file are "put bin" and "get bin". These procedures take two parameters: a TRANSIT variable, and a list of objects to be transferred, enclosed in brackets if there is more than one. When buffered transput is used the items transferred can have any of the basic modes CHAR, INT, LONG INT, BITS, REAL, BOOL, BYTES, LONG BYTES, LONG LONG BYTES or be structures composed from these modes. If the objects given to put bin are variables, they are automatically dereferenced, but no dereferencing takes place within structures, and references cannot be transferred. Arrays of items can be transferred bodily, as can structures containing arrays. When direct transput is used, put bin and get bin can only handle simple arrays (see section 4.2).

The transput procedures normally start at the beginning of the file when a channel is first opened and move through the file sequentially as items are transferred. If both input and output are being carried out on the same channel the procedures work independently; the current position or 'address' for putting and the current address for getting are held separately in the transit variable controlling the channel. The units in which the addresses are held depend on whether direct or buffered transput is being used. With direct transput the address is always a line number; line numbers start at 1 and go up to the total number of lines in the file. When a channel is opened the addresses for both putting and getting are set to line 1, and after each call of put bin or get bin the appropriate address is updated by the number of lines used, ready for the next transfer. (Remember that with direct transput, the data transferred in each call is made up with rubbish if necessary so that it always occupies a complete number of lines.) With buffered transput, the items being transferred are relatively small, and it is necessary to hold addresses in terms of characters (ie quarter words). Character addresses start at 1 and go up to the total number of characters in the file. Thus, with the standard line length of 512 characters, character addresses 513 - 1024 are all on line 2 of the file. Data items transferred using buffered transput take up only as many characters as necessary, the rest of the file being unaltered. After each call of put bin or get bin the appropriate address is updated by the number of characters used, ready for the next transfer.

Both put bin and get bin deliver an integer result which is the updated current address for putting or getting respectively. This integer is a character number if buffered transput is being used, and a line number if the transput is direct.

5.1 Random addressing

For a binary file or file on a private disc, the hardware used allows data to be accessed or stored anywhere in the file; it need not be dealt with merely as a continuous stream. Such random access is arranged in the program by resetting the current address for putting or getting as required, using the procedures described below; the next call of put bin or get bin will then transfer data starting at the new address.

The four procedures for address setting enable either the putting or getting address to be set in terms of line numbers or characters. The line setting procedures apply primarily to direct transput but may be useful with buffered transput for moving to an entirely different part of the file. The character setting procedures apply only to buffered transput. Since an item transferred using buffered transput takes up only as many characters as necessary and leaves the rest of the file unaltered, this makes it possible, for example, to set a character address for putting and write a single character in the middle of a file without disturbing its other contents. The address setting procedures are:

```
PROC setline p = (REF TRANSIT t, INT line number)
    sets the address for putting to line number
PROC setline g = (REF TRANSIT t, INT line number)
    sets the address for getting to line number
PROC setchar p = (REF TRANSIT t, INT char number)
    sets the address for putting to char number
PROC setchar g = (REF TRANSIT t, INT char number)
    sets the address for getting to char number
```

5.1.1 Simultaneous address setting and transput

The procedures "put to" and "get from" combine address setting and transput. They are similar to put bin and get bin, but have an additional integer parameter representing the address for the transfer. The integer is automatically interpreted as a line number if direct transput is being used, or as a character number for buffered transput. Thus:

<u>direct transput</u>	<u>equivalent</u>
openb(td, 1, direct);	
put to(td, 4, arr)	setline p(td, 4); put bin(td, arr)
get from(td, 4, arr)	setline g(td, 4); get bin(td, arr)
<u>buffered transput</u>	
openb(tb, 1, buffered);	
put to(tb, 81, (x,y))	setchar p(tb, 81); put bin(tb, (x,y))
get from(tb, 81, (x,y))	setchar g(tb, 81); get bin(tb, (x,y))

5.2 Serial addressing

If serial addressing is employed, data is written as a continuous stream, each call of an output procedure following on from the address where the previous call finished. This occurs naturally if all output is performed by put bin and no address setting procedures are called. To provide an additional check that all output addresses are in sequence the option "serial file" can be included in the list for openb (overriding the standard option "random file"). This monitors address setting for output and causes a fault if an address discontinuity occurs. It does not affect the addresses used for input.

5.3 Finding out the current address

There are four procedures for determining the current address for getting or putting. They are

PROC linenumber p = (REF TRANSIT t)INT

delivers the current line number for putting

PROC linenumber g = (REF TRANSIT t)INT

delivers the current line number for getting

PROC charnumber p = (REF TRANSIT t)INT

for buffered transput only, delivers the current address for putting as a character number

PROC charnumber g = (REF TRANSIT t)INT

for buffered transput only, delivers the current address for getting as a character number

In addition the procedures put to, get from, put bin and get bin all deliver the new current address as their integer result. With buffered transput this integer is the next character address for transfer, whilst with direct transput it is the next line number.

6 Groups

The objects transferred by a single call of "put to" or "put bin" are regarded as an indivisible group of data, often referred to as a record. When a group is output an additional word is automatically written at the head of the group; this header word contains the size of the group in characters, including the space taken by the header word itself. On recovery, the procedure get from or get bin checks that the group recorded at the address being used is not smaller than the group demanded. If it is smaller a fault occurs with the message BT-RETRIEVED GROUP IS WRONG SIZE. This is a valuable safeguard against the use of incorrect backing store addresses.

The options for openb which determine the grouping of data are "grouped" (the standard option) and "ungrouped", which arranges for data to be transferred without a header word. It must be emphasized that a group comprises the complete list of data objects transferred in a particular call of a transput procedure. This is shown in the following example, in which two transit variables tvg and tvu are opened using the grouped and ungrouped options respectively. The box diagrams show the effect of the header word, which contains the count in characters.

TRANSIT tvg, tvu;

```
openb(tvg, 1, buffered);
openb(tvu, 2, (ungrouped, buffered));
INT i := 1, j := 2, k := 3;
put bin(tvg, (i, j, k));
put bin(tvu, (i, j, k));
```

16	1	2	3
1	2	3	

```
put bin(tvg, i);
put bin(tvg, j);
put bin(tvg, k);
put bin(tvu, i);
put bin(tvu, j);
put bin(tvu, k);
```

8	1	8	2	8	3
---	---	---	---	---	---

1	2	3
---	---	---

The grouping of data applies to both buffered and direct transput. The header word must always be a complete word in the file; with buffered transput this is allowed for by put to, put bin, get from and get bin, which round up the current address to a word boundary after each group is transferred. Extra care should be taken if address setting procedures are being used. For direct transput, the header word is dealt with specially; it need not be allowed for in the array.

The size of a group is stored as a count of words rather than characters if the option "wordheader" is supplied to openb, overriding the standard option "charheader". The procedure "openarray" described in section 3 is equivalent to a call of openb with the options (direct, wordheader).

6.1 Skipping over groups in a serial file

When a file contains grouped data which has been output serially (see section 5.2) the recovery of a particular group may involve reading through the file until the required group is reached, since in general the starting address of a group is not known. This process can be speeded up by using one of the procedures

```
PROC jump g = (REF TRANSIT t, INT n)
PROC jump p = (REF TRANSIT t, INT n)
```

These procedures skip over the number of groups given by n (which must be positive) and set the current address for getting or putting respectively to the start of the next group. This is accomplished by inspecting the header word of each group in turn and incrementing the appropriate address accordingly. Since the contents of the group itself are not transferred the jump process will often be more efficient than recovering every group. As it involves getting one word of information the procedures cannot be used on a channel opened only for output.

7 Direction of transput; input, output or both?

The standard option taken by openb allows both input and output to occur on a single binary channel. One transit variable is capable of controlling both operations, since a TRANSIT is a structure with two fields, one called "put" and the other "get". The "get" field is itself a structure containing all the information pertinent to input on the channel, whilst the "put" field covers output.

If only one direction of transput is needed, the option "input" or "output" may be given. No output will be allowed on a channel opened for input, and vice versa. Since input and output involve distinct sets of library procedures, these options enable program space to be saved by allowing a limited number of library segments to be incorporated (see section 10).

There is a special output option "offset" corresponding to the ICL 'open at end of file' mode. For details of this facility, see the ICL direct access manual.

7.1 Common buffering for input and output

With buffered transput, the use of both directions of transput implies that buffer space for two lines must be kept, the current line for input and the current line for output. The buffers are automatically interconnected if they hold the same line of the file, so that an item may be read as if from the file even before the output buffer is transferred. (This does not apply between different transit variables - see section 8.) As soon as the input and output lines become distinct the buffers are again treated independently. Buffer space can be saved if it is known that the put and get fields of the transit will be working on the same line during most of the program. The option "common buffered" may be given to openb instead of "buffered". Space for only one line will then be taken, and this will be shared between the put and get fields of the transit. Should the input and output lines become distinct, an excessive amount of backing store transfer will take place as the contents of the buffer are changed from the current line for input to the current line for output. The saving which accrues from "common buffered" is purely a space saving of one line (normally 128 words).

8 Working with several transit variables

The linkage between a binary file and a transit variable is via a channel number, as described in section 2. Each binary file used must have its own channel and be associated with the channel number by a single George ASSIGN command. However the number of transit variables used to control the channel is not limited. Several transit variables can be associated with the same channel number by calls of openb, and thus work on the same binary file. This enables different methods of transput to be used for different parts of the file. For example, a transit variable initialised with the options buffered and ungrouped could be used to transput an index of data and addresses at the beginning of a file, whilst a second variable, initialised with openarray, could control the transput of the actual data.

When several distinct transit variables are linked to the same file, care must be taken to ensure that data is actually written to the file before it is requested by a different transit variable. With buffered transput, the contents of the buffer are only transferred to the file when the current address for output crosses a line boundary. The procedure

```
PROC dump buffer = (REF TRANSIT t)
```

may be used to force the contents of a buffer to be transferred.

When a channel is opened using the buffered option, space for a buffer area is taken from the data space allotted to the program. The size of the buffer corresponds to the size of a line in the file. Thus a standard line of 512 characters will require 128 words of data space per transit variable if any of the options input, output or offset is used, or 256 words per transit otherwise. This accumulation of space required must be allowed for in the STORE command given for the program.

9 Procedures using a previously selected transit variable

The following procedures assume continued use of the same transit variable and therefore have no transit variable parameter. Their identifiers are made up from those of the standard procedures with the prefix "now", and they have the same action.

```
now get bin ( variable list )  
now put bin ( value list )  
  
now get from ( address, variable list )  
now put to ( address, value list )  
  
PROC now setline g = (INT line number)  
PROC now setline p = (INT line number)  
PROC now setchar g = (INT char number)  
PROC now setchar p = (INT char number)  
  
PROC now jump g = (INT n)  
PROC now jump p = (INT n)
```

Each procedure uses either the currently selected "put" or the currently selected "get" transit field (section 7 explains the put and get fields). During binary transput a record is always kept of the get field of the transit variable last used for input, and the put field last used for output. These fields may come from distinct transit variables or from the same one; each time a transit variable is encountered as a parameter of an input or output procedure the currently selected get or put field is changed accordingly. (A procedure which does not involve input or output will change the current get field if the transit variable was initialised with "input", the put field if it was initialised with "output" or "offset" and both fields otherwise.) An explicit selection can be made by calling one of the procedures

```
PROC select get = (REF TRANSIT t)  
    selects the get field of the transit variable t,  
    leaving the currently selected put field  
  
PROC select put = (REF TRANSIT t)  
    selects the put field of the transit variable t,  
    leaving the currently selected get field  
  
PROC select both = (REF TRANSIT t)  
    selects both put and get fields of t
```

Thus, the call put bin (t, (i,j,k))
is equivalent to select put (t); now put bin((i,j,k))

10 Saving program space

The binary transput procedures are held in the system library in more than 20 different segments. Each segment performs a distinct operation, so that there are separate segments dealing with direct input, direct output, buffered input, buffered output and the grouping of data. When the option direct or buffered is given, all the segments dealing with that type of transput are incorporated into the program, since potentially any of them may be used.

Program space can be saved by restricting the segments incorporated to those that are actually needed. If, for example, all the transit variables in the program are used solely for input, only segments dealing with input will be needed. The restriction is achieved by replacing each occurrence of direct or buffered by a composite identifier, showing which direction of transput is in use (if there is only one) and, for buffered transput only, whether the data is ungrouped. The first step is to give an explicit input, output or offset option in each call of openb if only one direction of transput is required (see section 7). Thereafter, the following table shows the replacement necessary.

original combination of options	replacement combination of options for minimum size of program
<u>If only input segments are needed</u>	
direct, input	direct get, input
buffered, input	buffer get, input
buffered, input, ungrouped	buf get ungp, input, ungrouped
<u>If only output segments are needed</u>	
direct, {output offset	direct put, {output offset
buffered, {output offset	buffer put, {output offset
buffered, {output, ungrouped offset	buf put ungp, {output, ungrouped offset
<u>Buffered, in either direction but no grouping</u>	
buffered, ungrouped	buf ungp, ungrouped
common buffered, ungrouped	com buf ungp, ungrouped

Notice that it is only direct, buffered and common buffered that are replaced by the special options; the other options required must still be given, including any options not mentioned in this section.

11 Channel closing

A call of the procedure

```
PROC closeb = (REF TRANSIT t)
```

relinquishes the binary channel controlled by t, after ensuring that the contents of any output buffer have been transferred. Closing a channel may involve operating system overheads, and for this reason a channel should only be closed if it is essential that it be re-opened and attached to a different file during a single program run. The procedure closeb also performs the more esoteric function of updating the system control area on disc for a file opened using the "offset" option. For details of this facility see the ICL direct access manual.

12 Special disc procedures

The procedures in this section apply chiefly to files on a private disc, although they can also be used for binary filestore files.

12.1 Finding out a file name

When a channel is opened, the George operating system gives information to the program about the file attached to that channel. If the option "disc enquiry" is given to openb, this information is assigned to a library structure variable "disc details", declared as

```
STRUCT ( LONG LONG BYTES filename,  
        INT generation,  
        version,  
        serial no,  
        new gen,  
        line size,  
        retention ) disc details;
```

The version, serial no (ie cartridge serial number), new gen, line size and retention fields apply to a file on a private disc. The use of the field "line size" is described in section 12.2. The structure "disc details" will of course be overwritten on each subsequent use of disc enquiry.

As an example, the following program fragment will print the names of the files used on each run of the program:

```
BEGIN TRANSIT ta, tb;  
    openb(ta, 1, (direct, disc enquiry));  
    print(("CHANNEL 1 IS ", filename OF disc details, newline));  
    openb(tb, 2, (buffered, disc enquiry));  
    print(("CHANNEL 2 IS ", filename OF disc details, newline))
```

12.2 Using a non-standard bucket size

The standard length for a line (or in ICL terminology, a 'bucket') is 128 words. If a file has been set up with a larger line size, the option "disc enquiry" must be given to the call of openb for the channel connected to the file. The integer "line size OF disc details" will then contain the size of a line of the file in units of 128 words, and the buffer size and line numbering will be automatically set to correspond.

12.3 Altering the size of a file

The procedure to alter the size of a file is

```
PROC alter size = (REF TRANSIT t, []INT ds) INT
```

The first parameter must be a transit variable controlling the channel to which the file is connected. For the second parameter, one or two integers may be supplied:

ds[1] must give the change of file size required, in units of 1024 words. (It may be positive or negative.) Remember that the maximum size of a binary filestore file is 245 KWORDS.

ds[2] is only needed for a file on a private disc; it must give the serial number of the disc.

The procedure delivers as its result the new size of the file in units of 1024 words. The procedure

```
PROC file size = (REF TRANSIT t) INT
```

delivers the size of a file in units of 1024 words. When either procedure is used the qualifier OVERLAY must be given in the George command which connects the file to the channel controlled by t.

12.4 Renaming a file

A filestore file is best renamed by the George command RENAME (George 5.12). A file on a private disc can be renamed by the procedure

```
PROC rename disc = (REF TRANSIT t, LONG LONG BYTES fn, INT ng, nv)
```

where t is the transit variable controlling the channel to which the file is connected. The channel must allow output, ie a channel opened with the "input" option cannot be used.

fn is the new file name

ng is the new generation number

nv is the new version number

13 Faults and event procedures

If an error occurs during a binary transput procedure the program normally faults, displaying an appropriate error message. The program failure routine will then automatically print the values of the following four integers:

getline	the line number from the currently selected get field of a transit variable
getchannel	the channel number associated with the current get field
putline	the line number from the currently selected put field of a transit variable
putchannel	the channel number associated with the current put field

The 'event' mechanism can be used to intercept particular faults and in some cases to enable the program to continue after the error. Each transit variable holds two "event" procedures of mode PROC(INT)BOOL, one as part of the put field of the transit variable and the other as part of the get field. When an error (ie an event) occurs, the transput procedure detecting it obeys one of these procedures, supplying as its actual parameter an integer denoting the nature of the event. A list of these integers is given in section 13.2. It is divided into two parts, recoverable events and irrecoverable events. When an irrecoverable event occurs the program obeys the event procedure and then faults with the message shown. After a recoverable event the result delivered by the event procedure is inspected and the fault routine is called only if it is FALSE. If the event procedure delivers TRUE, implying that the error has been satisfactorily dealt with by the event procedure itself, the program continues, bypassing the fault completely. The standard settings for the event procedures and the way in which they can be changed is described in section 13.1.

Since each transit variable holds an event procedure in both its put and get fields it must be made clear which one is obeyed in any given situation. If the event occurs within a transput procedure which includes the word put or the postfix p in its identifier then event OF put OF the transit variable is called. If the procedure includes the word get or the postfix g in its identifier then event OF get OF the transit variable is called. The magnetic tape procedures "skid" and "reset line" also call event OF get OF the transit variable. For all other transput procedures the choice depends on the options supplied to openb for the transit variable. If these included "input" then the get event is called, otherwise the put event is called.

13.1 The standard event settings and how to change them

The setting of the event fields of both put and get parts of a transit variable is performed by the following fragment of the library procedure openb:-

```
PROC nul event = (INT e)BOOL:  
    { IF e = 1001 AND headersize >= groupsize  
      THEN TRUE  
      ELSE FALSE FI );  
  
event OF get OF transit variable := nul event;  
event OF put OF transit variable := (INT e)BOOL: (FALSE)
```

In every case but one these procedures deliver FALSE, thus causing a fault. The exception is event number 1001, which occurs during grouped transput when the size of the group demanded ("groupsize") is different from the size of group recorded on backing store ("headersize"). The identifiers groupsize and headersize are library integers which have the appropriate values assigned to them before the event procedure is called. The procedure compares the two integers and allows the program to continue if the group requested is smaller than that originally recorded. If it is larger the procedure delivers FALSE and a fault will occur. As well as groupsize and headersize there is one other library integer, "eventdata", which provides additional information about certain events; details can be found in the notes on the events to which it applies.

Either of the event procedures can be changed by assigning a new procedure to event OF put or event OF get of the transit variable, overriding the initial assignment above. The new procedure will then be obeyed when an event occurs, the choice between a call of the put event and a call of the get event being given in section 13. The procedure can be used to perform extra actions before an irrecoverable event (eg printing eventdata) and to recover from other events by delivering TRUE instead of FALSE. The assignment(s) should normally be made after the call of openb, but if the event to be trapped is likely to occur in openb itself then assignments to both put and get fields must be made and the option "eventset" included in the list for openb. The standard assignments shown above will not then be carried out.

The following are examples of assignments to event OF get of a transit variable. The first adds to the standard response described above, while the second overrules it, causing the program to fault if the size of the recorded group does not equal the size of group requested.

- (1) event OF get OF transit variable :=
(INT e)BOOL: (IF e = 2000 THEN print (event data); FALSE
ELSE nulevent(e)
FI)
- (2) event OF get OF transit variable := (INT e)BOOL: (FALSE)

13.2 Event numbers and associated error messages

recoverable events

event number	error message
1001*	bt-retrieved group is wrong size
1002	bt-skid after write not allowed
1003	bt-tape cannot be reset at the start
1004	bt-slice is not a contiguous array
1005	bt-direct option only handles arrays
1006	bt-address discontinuity with serial
1007*	bt-only one item at a time with direct
1008	bt-address discontinuity with tape
1009	bt-disc groups must start at a word
1010	bt-tape line larger than get demands
1011**	bt-tape mark detected
1012	bt-too many reverse skids demanded
1013*	bt-not enough room for filesize demanded
1014†	bt-tape line smaller than get demands

irrecoverable events

event number	error message
2000*	bt-executive report
2001	bt-no put with this transit variable
2002	bt-no get with this transit variable
2003	bt-open error;direct or buffered?
2004	bt-common buffered does not apply
2005	bt-only use this proc for disc
2006	bt-only use jump when input allowed
2007*	bt-executive report during disc enquiry
2008	bt-executive report during tape enquiry
2009	bt-only use this proc for tape
2010	bt-line and char counts start at one
2011	bt-only use jump with grouped
2012	bt-you've gone over the end of the tape
2013	bt-you've gone over the end of the file
2014	bt-this proc needs an overlay channel
2015	bt-skid goes beyond last valid block
2016	bt-group too large for one tape buffer

* See special note in section 13.3

** See special note in section 14.2.1.1

† See section 14.1

LIBRARY
Binary Transput
13.3

13.3 Notes on particular events

Event 1001

This event occurs during grouped input whenever the size of group demanded is different from the size of group recorded. Before the event procedure is called, the size of the group as recorded on backing store is assigned to the library integer "headersize", and the size of the group demanded is assigned to the integer "groupsize", expressed in words or characters depending on the option given to openb. An amount of data equal to "groupsize" will have been transferred when the event procedure is called. Note also section 13.1.

Event 1007

This event applies only to direct transput and occurs when more than one array is given in a single call of putbin, getbin, put to or get from. The event procedure is called after the transput of all the arrays has been performed; for grouped transput each array is treated as a separate group. The total number of arrays transferred is assigned to the library integer "eventdata".

Event 1013

Before the event procedure is called the maximum number of ICL blocks available is assigned to the library integer "eventdata". Note that 1 ICL block is equal to 1 line ('bucket') for a file created with BUCKET1.

Events 2000 and 2007

Ultimately most transput operations involve a call to the ICL Executive system. If the operation fails, Executive returns a 'reply word' to the program with certain bits set depending on the cause of the failure. The common cases are automatically decoded by the binary transput routines and a suitable error message is printed (for example, YOUVE GONE OVER THE END OF THE FILE is one of these). The less common cases simply assign the reply word to the library integer "eventdata". Event 2000 occurs after a transfer failure or failure in rename disc, file size or alter size; event 2007 occurs after a failure to open a file.

13.4 Transput within an event procedure and warning on scopes

Extra binary transput may be carried out inside an event procedure but it is the users responsibility to ensure that the currently selected put and get transit fields are the same on exit from the procedure as they were on entry (see section 9). In addition, as an event procedure may be called at any time during binary transput it is essential to ensure that every identifier or label used by the procedure is in scope whenever transput is taking place.

14 Magnetic tape

In principle, the binary transput procedures can be used to transfer data to magnetic tape simply by opening a tape channel, as described in 14.1. However, the physical characteristics of magnetic tape affect the way in which the procedures can be used. There are two main points. Firstly, a tape is essentially serial; data is read or written as the tape moves forward past a fixed recording head. Consequently, transput can only be carried out by "put bin" and "get bin", and random address setting is not allowed. This is discussed further in section 14.2. Secondly, data on tape is recorded in variable length blocks, separated by an inter-block gap. The size of a block is determined by the amount of data written on the tape in a single hardware transfer. This means that the position of any particular item of data depends on what has been written before it, and the output of a block of data makes any data previously recorded beyond the writing point inaccessible. For this reason a tape is normally used only for input, or only for output. This is discussed further in section 14.1.2.

It should be made clear at the outset that the Algol 68-R procedures for tape handling provide ways of performing the basic transput operations. With the single exception of the procedure "end file" they do not automatically take account of the extensive ICL conventions for data on magnetic tape. Users who find it essential to observe the ICL house-keeping conventions should refer to the ICL manual 'Magnetic Tape', which gives details of the formats required.

14.1 Opening a magnetic tape channel

A magnetic tape channel is opened by including the identifier "tape" in the list of options given to the procedure "openb" (described in section 4). The channel must be connected to the required magnetic tape by an appropriate George command in the job description. The other sets of options for openb which must be considered are those concerned with the grouping of data (14.1.1), the direction of transput (14.1.2) and the buffering arrangements. A choice must always be made between "direct" and "buffered" transput (sections 4.1 and 4.2). For magnetic tape this choice will also determine the size of blocks written to the tape. When direct transput is used, each call of put bin performs a hardware transfer, and thus writes a block equal in size to the array, plus one word for the header word if the transput is grouped. For buffered transput, a hardware transfer is automatically performed whenever the buffer is full, thus giving regular blocks equal in size to the buffer. This is normally 128 words, but can be changed by including the option "tapelinesize(s)" in the list for openb, where s is the size of buffer required, in words. Thus

```
openb (tv, 1, (tape, tapelinesize(200), buffered))
```

would set the buffer size to 200 words. The user can also force a hardware transfer to take place before a buffer is full by calling the procedure "put line" described in 14.1.1, thus writing a shorter block than normal. On input, buffered transput will cope automatically with blocks of variable length. The buffer size should be set to the maximum size of block expected; if a larger block than this is encountered, event 1010 occurs (see section 13). If a smaller block is read, the amount of valid data transferred is remembered, and the next block automatically read in when this data is exhausted. With direct transput, event 1010 occurs if a block larger than the array (plus header word for grouped transput) is read. If the block is smaller, the number of words transferred is assigned to "eventdata" and event 1014 occurs.

14.1.1 Grouping of data on magnetic tape

If no special option is supplied to openb, data is written or read in 'groups'. This applies for both direct and buffered transput, and is described in section 6. There is one restriction that applies to magnetic tape. When buffered grouped transput is used, a group must not be split across two blocks; each group output must be contained entirely within the current buffer. This is because the size of a group is calculated after each item in the group has been transferred by "put bin". If the start of the group is in a block already written to tape the procedure cannot return to the header word to transfer this information. When the output data consists of regular cycles of groups of known length, this problem can be avoided by adjusting the buffer size (using "tapelinesize") so that a whole number of groups will exactly fit. Alternatively, when it is known that the next group to be output is too large to fit into the remainder of the buffer the procedure

```
PROC put line = (REF TRANSIT t)
```

can be called. This will transfer that part of the buffer used so far to tape, thus giving a shorter block than normal. (Note that a block on magnetic tape must be at least 5 words.) The next group output will then start a new buffer. The "put bin" procedure can only do this automatically for a group containing a single item of data which occupies a contiguous area of store, for example a structure not containing an array, or a contiguous part of an array.

14.1.2 Direction of transput on magnetic tape

The output of a block of data to magnetic tape makes any data previously recorded beyond the writing point inaccessible. For this reason a tape is normally used solely for output, when the option "output" should be given to openb, or solely for input, when the option "input" should be given. The use of these options will prevent any output operations on a channel opened for input and vice versa. It also enables program space to be saved as described in section 10.

A combination of input and output is needed if data is to be appended to a tape, ie the tape is to be read up to the end of the information recorded on it and then extra data is to be written. This presents some difficulty, as the tape must be accurately positioned before writing starts. A way of doing this is described in 14.3, but it may be better to use two separate tapes and avoid the problem altogether.

The standard option taken by openb permits both input and output. When buffered transput is used, space for only one buffer will be taken as the input and output positions can never be distinct.

The option "offset" does not apply to magnetic tape.

14.2 Accessing magnetic tape

Access to magnetic tape is governed by the physical position of the tape, since data can only be read or written as the tape passes a fixed recording head. Movement of the tape is initiated by the transfer of data. Thus, when a block is output, data is transferred to the current tape position and the tape is moved along accordingly. Similarly on input, data is recovered starting at the current tape position, and the tape is moved along. To recover a particular item of data the tape must be positioned at the point where the data is recorded. This can be done by reading through the tape up to that point, using calls of "get bin". Alternatively, if the data is grouped and the input is buffered, the procedure "jumpg" described in 6.1 may be used. In this case, although all the data on the tape will be read it will not be transferred from the buffer to store variables. Faster movement of the tape (in either direction) can be achieved by the use of 'tape marks' as described in 14.2.1.

14.2.1 Tape marks

A tape mark is a single row of data (not a block) which can be recognised by the tape handling hardware. The hardware can be instructed to move the tape forwards or backwards until the next tape mark is encountered. This facility makes it easier to position the tape at any point. A tape mark is output by calling one of the procedures

PROC mark = (REF TRANSIT t)VOID

PROC now mark = VOID {this uses the currently selected transit variable as defined in section 9}

When buffered transput is used the contents of the buffer are transferred before the mark is written. On input the corresponding procedures

PROC skid = (REF TRANSIT t, INT n)

PROC now skid = (INT n)

will move the tape past n tape marks, forward if n is positive and backward if it is negative.

After a tape mark it is customary (but not essential) to output a special block of data to identify the tape mark in some way. This makes it possible to recognize a specified tape mark by program, skipping to each mark in turn and inspecting the following block of data. It is particularly useful for finding the end of the information recorded on the tape.

14.2.1.1 The tape mark event

If a tape mark is encountered when an input procedure is attempting to read the next block of data, the "event" procedure of the get field of the transit variable is called with 1011 as its actual parameter. This happens both for the explicit transfer carried out by each call of get bin when direct transput is being used, and for any implicit transfers performed by buffered transput. The general 'event' mechanism is described in section 13. For event 1011, if the event procedure delivers FALSE the program faults with the error message BT-TAPE MARK DETECTED. If the event procedure is changed to deliver TRUE, the transfer is repeated and the program continues. This is shown in the following example, which reads 100 blocks of data, ignoring any tape marks:

```
TRANSIT tv; openb(tv,1,(direct,tape,ungrouped));  
event OF get OF tv :=  
    (INT e)BOOL: (IF e=1011 THEN TRUE ELSE FALSE FI);  
    [1:100, 1:400] INT a;  
FOR i TO 100 DO getbin (tv, a[i])
```

14.2.2 Addresses on magnetic tape

The concept of an 'address' for output and a separate 'address' for input does not apply to magnetic tape. The parts of the controlling transit variable which normally hold these addresses are used for a different purpose. The address part of the get field holds the current position of the tape, while that of the put field gives the last position on the tape used for output, ie the end of the valid information written during the current program run. The positions are kept in terms of line numbers (a 'line' is a block of data) and tape mark numbers. Initially both counts are set to 1. Thereafter the line number in the get part of the transit variable is incremented by 1 every time a block is transferred to tape or read. When a tape mark is encountered or written the tape mark number is incremented and the line number is set back to 1. A skid backwards over n tape marks decreases the tape mark number by n, and sets the line number to zero, since the actual line number relative to the previous tape mark is not known. On output, the position in the put field of the transit variable is updated from the current position in the get part. There are four procedures for determining the position from the get or put part of the transit variable. They are (with the obvious significance)

```
PROC line number g = (REF TRANSIT t)INT  
PROC tm number g = (REF TRANSIT t)INT  
PROC line number p = (REF TRANSIT t)INT  
PROC tm number p = (REF TRANSIT t)INT
```

Note that if two transit variables are used to control the same tape channel their counts are incremented independently and will not necessarily tally with the tape position.

14.2.3 Special tape positioning procedures

A tape can be moved backwards one block by calling one of the procedures

PROC reset line = (REF TRANSIT t)VOID

PROC now reset line = VOID {this uses the currently selected transit variable as defined in section 9}

The line number in the get part of the transit variable is then decreased by 1. It is not possible to output data immediately after backspacing, and it is inadvisable to backspace consecutively a large number of times as there is a danger that the tape may become progressively out of step with the recorded blocks. The procedure is most useful for recovering after event 1010 (when a block is longer than expected) and when appending data.

The procedure

PROC rewind = (REF TRANSIT t)

will rewind a tape to its beginning, so that it can be used again in the same program run. There is a difference of one block between the position of the tape after rewinding and its position at the start of a program run. The first block recorded on every tape is known as a header label and contains the tape name and serial number. When a tape is connected to a channel this header label is read by George, to ensure that the correct tape has been loaded, and the tape is thus positioned after the header label. Rewinding positions the tape before the header label, and for consistency sets the line number to zero so that this first special block can be read by the program before processing starts. The header label must not be overwritten; this can only be done by the special procedure "rename tape".

14.3 Writing after reading

Writing after reading is only necessary when a tape is to be made up by several program runs, each one adding another batch of data. At the start of each run the tape must be positioned accurately after the previously recorded information. This is usually done by the following process. The procedure

PROC endfile = (REF TRANSIT t)

is called at the end of a run to terminate the data written. This outputs a tape mark followed by a 20 word block with its first word set to 7. On the next program run the procedure skid is used to position the tape past this mark and the following block is read to check that the correct tape mark has been found. The procedure skid is called again with -1 as its integer parameter, followed by a call of "reset line". This positions the tape before the last block of data required. This last block must then be read, so that the tape is correctly positioned after it for the new information to be added (overwriting the tape mark and terminating block). It is not possible to write immediately after a backwards movement of the tape. More general methods are described in the ICL manual 'Magnetic Tape'.

LIBRARY
Binary Transput
14.4

14.4 Tape names

Each magnetic tape has a name and a serial number, and these are recorded at the beginning of the tape in a special data block called a header label. The header label also contains various other details of the tape. When a tape is connected to a channel by a George command, its header label is read and checked by George. If the special option "tape enquiry" is given to openb, the information obtained by George is passed to the program and assigned to the library structure variable "tape details", which has the form

```
STRUCT (LONG LONG BYTES file name,  
        INT serial no,  
        mode of online,  
        reel seq no,  
        generation,  
        retention,  
        date written) tape details;
```

The structure will of course be overwritten on each subsequent use of tape enquiry.

Changing the name of a tape involves overwriting its header label, and this can only be done by the special procedure

```
PROC rename tape = (REF TRANSIT t, LONG LONG BYTES fn,  
                    INT ng, nsq, nrt)
```

where fn is the new file name (ie tape name)
ng is the new generation number
nsq is the new reel sequence number
nrt is the new retention period

The procedure rewinds the tape to the very beginning and rewrites the header label. Any data previously recorded on the tape is lost.

14.5 The end of the tape

If a tape is being used for output, the procedure

```
PROC end file = (REF TRANSIT t)
```

should be called after the last call of "put bin" on the tape channel. When buffered transput is in use this ensures that the current buffer is transferred. For both buffered and direct transput the procedure then outputs a tape mark followed by a 20 word block of data with its first word set to 7 and the remaining words indeterminate. This is the ICL convention for an 'end of composite file' marker, and makes it easier to add further data to the tape on a subsequent run. To relinquish the channel completely the procedure

```
PROC closeb = (REF TRANSIT t)
```

can be called instead (it includes a call of "end file"), but this is not normally necessary.

The physical end of a tape is marked by a metallic strip some way before the end of the recording surface. If this is encountered when data is being output, the transfer is completed and the event procedure of the put field of the transit variable is then obeyed with parameter 2012 (see section 13). At this point a recognisable data block should be output to denote the end of information on the tape; this is necessary because the end of tape marker cannot be detected on input. Note that once the marker has been reached, the event procedure will be recalled with the same parameter on all subsequent data transfers, although the transfers themselves will be completed successfully.

15 Summary of options for openb

The standard option taken in each set of options is marked with an asterisk. Section numbers are given in square brackets.

Buffering arrangements

direct	[4.2]	Note: one option from this set must always be given
buffered	[4.1]	
common buffered	[7.1]	

Grouping of data

*grouped	*charheader	[6]
wordheader		

ungrouped

Direction of transput

*overlay	(ie both directions)
input	[7, 14.1.2 (tape)]
output	[7, 14.1.2 (tape)]
offset	[7]

Backing store device

*disc	
tape	[14.1]

Type of access (applies to disc only)

*random file	[5.1]
serial file	[5.2]

Miscellaneous options (use as required)

eventset	[13.1]
disc enquiry	[12.1]
tape enquiry	[14.4]
tapelinesize(s)	[14.1]

Replacement options for minimising program size

See section 10.

CONSTANTS

The following identifiers, of mode REAL, represent commonly occurring constants

REAL halfpi	=	1.5707 9632 679
REAL infinity	=	5.7896 0446 188 & 76
REAL log10e	=	0.4342 9448 1903
REAL pi	=	3.1415 9265 359
REAL root2	=	1.4142 1356 237
REAL twopi	=	6.2831 8530 718

The following identifiers, of mode LONG LONG BYTES, give the name of the job and the username (without an initial colon) under which the program is running. The LONG LONG BYTES value in each case is the name, left-justified and space-filled if necessary.

LONG LONG BYTES	jobname
LONG LONG BYTES	user



INTEGRATION OF ORDINARY DIFFERENTIAL EQUATIONS
(RUNGE-KUTTA-MERSON)

```
PROC autograde = ( PROC(REF[ ]REAL,REF[ ]REAL)BITS auxiliary,
                    REF[ ]REAL solution,
                    REAL end,
                    REF REAL step,
                    REALPAIR accuracy )
```

This is a general procedure for step-by-step solution of differential equations. The equation or equations may be of any order and need not be linear, but must first be re-cast as a set of first-order simultaneous equations:

$$\begin{aligned} \frac{dy_0}{dx} &= 1.0 \\ \frac{dy_1}{dx} &= f_1(y_0, y_1, \dots, y_n) \\ &\dots \\ \frac{dy_n}{dx} &= f_n(y_0, y_1, \dots, y_n) \end{aligned}$$

The first equation enables the independent variable x to be treated as a dependent variable y_0 , the value of which is maintained by autograde as an integration of increments to x . The most up-to-date values of this and of all the other y's are kept in the array variable "solution". The size of this array determines the number of equations to be solved, and the bounds of the array may be chosen arbitrarily. For simplicity, they may be taken as 0 to n . The differential equations must be programmed as an auxiliary procedure which is called automatically within autograde whenever it needs a fresh set of derivatives for trial values of the y 's. Autograde supplies y 's of its own choosing in the first parameter of the auxiliary, which must be written to compute the corresponding f 's and place them in its second parameter. (The 1.0 need not be put in the bottom array element, as this particular f is set automatically by autograde before the auxiliary is first called.) The bounds of the two array parameters of the auxiliary are the same as those chosen for the array variable "solution". The auxiliary should normally be designed to deliver the value BIN 0, ie the BITS equivalent of the integer zero. If the last unitary clause obeyed in the auxiliary is written as 0, conversion to BITS is automatic.

[continued]

Example of auxiliary for autograde

The equation for the angular velocity of a hunting governor is:

$$y''' - y'' + 2y = 40$$

and by writing y_1 in place of y , this may be cast in the form:

$$y_0' = 1.0$$

$$y_1' = y_2$$

$$y_2' = y_3$$

$$y_3' = y_3 - 2y_1 + 40$$

A skeleton auxiliary for this problem is:

```
PROC governor = (REF[ ]REAL y,f)BITS:  
BEGIN f[1] := y[2];  
      f[2] := y[3];  
      f[3] := y[3] - 2*y[1] + 40;  
      0  
END;
```

Call of autograde

When the parameters of autograde have been set up, including starting values in "solution", one call of autograde initiates a complete run of integration. The step sizes within the run are adjusted automatically, being maximized to ensure that the relative or absolute errors in the solution are just within the limits specified in the "accuracy" parameter (see Approximation 1), but a guessed value of "step" must be supplied to start things off. As there is no exit from autograde until the independent variable (solution[0] if the lower bound is 0) reaches the value "end", running values of the solution must be inspected, if they are required, from within the auxiliary procedure. This is entered several times between each up-dating of solution; to prevent repetition when printing out solutions, it is necessary to test whether the independent variable has been stepped on since it was last used. This is shown in the following full example.

Full example

```

BEGIN [0:3]REAL solution := (0.0, 1.0, 0.0, 0.0)[AT 0];
    REAL step := 0.1, end := 10.0;
    REAL last x := solution[0] - step;
    REALPAIR rough := (1&-3, 1&-3);
    after OF numberstyle := 3;

    PROC hunter = (REF[ ]REAL y, f)BITS:
        BEGIN IF (last x - solution[0])*step < 0
            THEN print ((solution[0:1], newline));
            last x := solution[0]
        FI;
        f[1] := y[2];
        f[2] := y[3];
        f[3] := y[3] - 2*y[1] + 40;
        O
    END;

    autograde (hunter, solution, end, step, rough)
END

```

Refinements

The variation in step size can be prevented by delivering suitable BITS values from the auxiliary. The BITS values can be expressed as integers:

- 0 allows normal adaptation
- 1 prevents reduction of step size, often useful at the start of a run
- 2 prevents increase of step size
- 3 prevents any change of step size

As an example, one might prevent excessive automatic refinement of step size in difficult but unimportant parts of the range of integration by concluding the auxiliary with the clause

IF step < so much THEN 1 ELSE 0 FI

[continued]

Advice

Autograde should not be used to integrate through a discontinuity in the variables or their derivatives. Use two calls, one up to the discontinuity, the second beyond it.

Although it is usual to take the mathematically independent variable as the independent variable for autograde, this is not necessary. It is possible to proceed in steps in y , and hence terminate the integration when the solution reaches a given value. For example, consider the equation

$$\frac{d^2y}{dx^2} + y = 0$$

Writing y as y_0 and x as y_1 , this may be cast as

$$dy_0/dy_0 = 1.0$$

$$dy_1/dy_0 = 1.0/y_2$$

$$dy_2/dy_0 = -y_0/y_2$$

which causes the solution to be obtained by stepping y instead of x .

FITTING AND LEAST SQUARES ANALYSIS

The problem is to choose the parameters in a mathematical equation to make its curve or surface pass through given points. If there are as many parameters as there are points, the fit can be exact (though when iteration is necessary it saves computing time if an approximate fit can be tolerated). When there are more points than there are parameters, an exact fit is impossible, as the points are presumably subject to experimental errors. However, the larger the number of points, the sharper will be the values obtained for the parameters. Various methods and library procedures are available to help with fitting, and it is important to select the simplest possible. The use of a complicated method can consume too much computer time, and even then fail to give any satisfactory result.

SIMPLE CURVE-FITTING (polyfit)

A curve is the plot of a function of one independent variable, t say, with parameters, X , controlling its shape:

$$(1) \quad y = f(t, X_0, X_1, X_2, \dots)$$

Given a set of points (y, t) , the problem is to find the best values for the X 's. Typical forms of f , which may be taken to any number of terms, are:

$$(2) \quad y = X_0 + X_1 \cos(t) + X_2 \cos(2t) + \dots$$

$$(3) \quad y = X_0 + X_1 t + X_2 t^2 + \dots$$

In such simple cases, f is linear in the X 's, ie

$$(4) \quad y = X_0 p_0(t) + X_1 p_1(t) + X_2 p_2(t) + \dots$$

If the nature of the problem allows a free choice, linear forms are always to be preferred, and the next thing to decide is what p -curves to take. Ideally, these should be a set of functions which are mutually orthogonal over the given range of values of t , as the coefficients X can then be obtained by well-known analytical methods. Failing an analytical approach, the p -curves should be chosen to facilitate numerical computation. Arbitrarily chosen curves are apt to lead to ill-conditioned equations and inaccurate parameters. This is the case for ordinary polynomial fitting, shown at (3), unless the calculation is properly designed.

The procedure polyfit should give good results, as it transforms the simple powers of t into orthogonal polynomials and back again. It computes the terms in (4) in sequence, and the process can be stopped when a good enough fit to the given y's is obtained. The procedure gives an exact fit if the number of points is no greater than the number of terms required in the polynomial.

GENERAL LINEAR FITTING (simeq, linear fit)

Curve-fitting is a special case where the fitting function is a function of one independent variable, t. More generally, we may wish to fit y as a function of several variables, $x_1, x_2 \dots$ with adjustable parameters $X_1, X_2 \dots$. For simplicity, this function should be chosen to be linear in the X's, eg

$$(5) \quad y = X_1 x_1 + X_2 x_2 + \dots + X_n x_n$$

Each point to be fitted is supplied as a multiple-value ($y, x_1 \dots x_n$). If n such points or multiple-values are given, the n substitutions in (5) give n simultaneous equations which can be solved for the X's using the procedure simeq (see Matrices 2). If more than n points are given, the procedure linear fit will minimize the mean squared error in the y's (assuming the x's to be exact).

The fact that (5) is written as linear in y and in the x's is irrelevant. In solving for the X's, the y's and x's are treated as given constants. They can in fact be any functions of observed physical quantities, linear or non-linear.* Points can be given different weights by substituting ($w_y, w_{x_1} \dots w_{x_n}$) into (5), with different values of w for each point.

* As an example, consider determining g by fitting the equation

$$s = vt + 0.5gt^2$$

to one or more measurements of (s, v, t). The equation has the form

$$y = Xx$$

where $y = (s-vt)$, $X = g$ and $x = 0.5t^2$.

NON-LINEAR PROBLEMS (inverlate, newton, secsol)

When the equation to be fitted to the given points is non-linear in the parameters we wish to determine, there is usually no simple and direct method of solution. If there are m points to be fitted, we find that we have m non-linear equations

$$(6) \quad F_i(x_1, x_2 \dots x_n) = 0 \quad [i \text{ to } m]$$

to be solved for the X 's, one equation for each given point. If $n = m = 1$, ie one equation $F(X) = 0$, the simplest solution is to use the procedure inverlate (see Interpolation 4). If $n = m$, greater than 1, the simultaneous non-linear equations can be solved exactly using newton or secsol. Of these, newton is preferable, but must be given the n partial derivatives for each of the n functions with respect to the unknowns. If these derivatives are not available, use secsol, which estimates the derivatives for itself.

When there are more points than parameters ($m > n$), the usual requirement is to minimize

$$(7) \quad \sum_{i=1}^m (F_i(x_1, x_2 \dots x_n))^2$$

This is performed by the procedure minsumsq, though if derivatives are obtainable analytically, the use of more efficient methods from the literature should be considered.

GENERAL LEAST SQUARES ANALYSIS (THEORY)

In all the methods described above, the fitting criterion is to minimize the sum of squares of the residuals ϵ_i in m equations of the form

$$(8) \quad F_i(x_1 \dots x_n, x_1 \dots x_k) = \epsilon_i$$

In practice, this is usually equivalent to assuming that only one of the x 's is subject to error. For example, the y in (5) can be regarded as one of the x 's in (8), and the theory assumed this to be the variable in error. In a more general statistical treatment, the "observables" x each have their own variance, described in terms of a covariance matrix W^{-1} . The general approach may be summarised as follows.

First linearize the equation $F=0$ to the following sum of m -component column vectors:

$$F^0 + D\delta + E\epsilon = 0$$

where

$$F_i^0 = F_i(x_1^0, \dots, x_n^0, x_1^0, \dots, x_k^0)$$

$$D_{ir} = \frac{\partial F_i}{\partial x_r}$$

$$E_{ij} = \frac{\partial F_i}{\partial x_j}$$

$$x_r = x_r^0 + \delta_r$$

$$x_j = x_j^0 + \epsilon_j$$

The x_r^0 are guessed values of x_r for iteration purposes. The x_j^0 are values of the x_j which would be exactly fitted by the function, and the ϵ_j are therefore a measure of the misfit in each observable separately. The criterion for fitting is to minimize $\epsilon^T W \epsilon$, where W^{-1} is the covariance matrix of the x 's. Provided that $E W^{-1} E^T$ is non-singular, it can be shown that

$$N\delta + CF = 0$$

$$\text{where } N = D^T (E W^{-1} E^T)^{-1} D$$

$$\text{and } C = D^T (E W^{-1} E^T)^{-1}$$

This is a set of n linear equations in the n unknowns δ_r which can be solved by using simeq. The whole solution is iterated by taking the x_r to be the new x_r^0 . Iteration can be stopped when the mean-

squared error fails to decrease significantly. This error is given by

$$\epsilon^T W \epsilon = F^T (E W^{-1} E^T)^{-1} F - \delta^T C F$$

If, as a special case, $F_i(x_1, \dots, x_n, x_1, \dots, x_k)$ is of the form

$$x_i = G_i(x_1, \dots, x_n)$$

and $W = I$, the error is simply

$$\epsilon^T \epsilon = F^T F - \delta^T C F$$

The second term gives the reduction of error achieved by the latest iteration. The covariance matrix of the solutions x_r is N^{-1} .

FUNCTION MINIMISATION

The most difficult numerical process is to minimize an arbitrary function of n variables with no derivatives available. It is a counsel of despair to tackle a fitting problem in this way. Although procedures may become available, they should be used only as a last resort.

FITTING (x, y) POINTS WITH GIVEN ORDER POLYNOMIAL

```
PROC polyfit = (REF[ ]STRUCT(REAL x, y) points,  
                REF[ ]REAL coef ) INT
```

The procedure constructs a polynomial of order n in x,

$$p_n(x) = c_0 + c_1x + c_2x^2 + \dots + c_nx^n$$

so as to minimize the sum of squares of $y - p_n(x)$ over a given set of (x, y) points. The procedure places the coefficients c_0 to c_n in the array variable coef, from the element of lowest index to that of highest. The order n of the polynomial is determined by the size of the array coef, ie UPB coef - LWB coef. The integer result of polyfit is UPB coef. The original array of points is overwritten by polyfit; the y-fields of the array elements end up holding the residuals $y - p_n(x)$.

To carry out a progressive fit, testing at each stage to see if the order of polynomial is high enough, use the more general procedure polylimit.

Inclusion of weights

The procedure polyfit will alternatively accept a "points" parameter of mode

```
REF[ ]STRUCT(REAL x, y, z)
```

When the actual parameter has this mode, the z-field of each array element must contain a weight to be associated with the (x, y) point. The procedure then minimizes the sum of squares of $(y - p_n(x)) * z$ over the given set of points. Finally the y-fields will contain these weighted (unsquared) residuals.

FITTING (x, y) POINTS WITH POLYNOMIAL OF ADAPTABLE ORDER

```
PROC polylimit = (REF[ ]STRUCT(REAL x, y) points,  
                  REF[ ]REAL coef,  
                  PROC(REAL)BOOL continue) INT
```

This procedure is a generalization of polyfit (q.v.) with an extra parameter enabling the user to adapt the order of polynomial so as to attain any desired degree of fit. The fitting proceeds in stages, first to obtain the best polynomial of order zero, then of order one, then two etc, up to the maximum order permitted by the size of the array coef. After each stage, polylimit calls the user's procedure "continue", and if this delivers TRUE, polylimit proceeds to the next order of polynomial. Otherwise it stops, and delivers as its result LWB coef + (order of polynomial reached), which is the largest index of coef used.

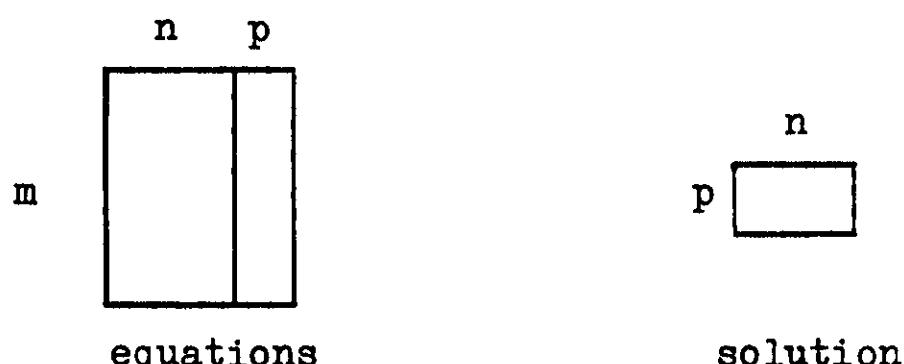
When writing the procedure "continue", any available information may be used to decide whether the fitting process should continue to a higher order or be terminated. For example, the y-fields of "points" contain the latest residuals. The sum of squares of these residuals will get smaller with each succeeding iteration, by R^2 say. The REAL parameter of continue (supplied by polylimit each time it calls continue) gives the value of R for the previous iteration. For a perfect fit, therefore, the sum of squares of the successive R's will equal the sum of squares of the given y's.

Weights can be attached to the given (x, y) points in the same way as for polyfit.

LINEAR LEAST SQUARES FITTING

```
PROC linearfit = (REF[,]REAL equations, REF[,]REAL solution)
```

This procedure does a least squares fit to find n unknowns from m linear equations, where the number of equations is greater than the number of unknowns. A single call of the procedure can be used to find solutions for p different values of the right-hand sides of the equations.



The parameter "equations" must be supplied as an m by $(n+p)$ matrix containing the coefficients of the equations (probably obtained from observations) in the first n columns. The one or more different sets of right hand sides should be placed in the remaining p columns and a p by n matrix variable "solution" provided to hold the results. The procedure will assign to each row of "solution" the solution for the corresponding column of right hand sides of the equations. If there is only one set of right hand sides the actual parameter for "solution" can be a one-dimensional array variable.

The array "equations" will not be altered by the procedure.

SOLUTION OF NON-LINEAR SIMULTANEOUS EQUATIONS
WITH COMPUTABLE DERIVATIVES

PROC newton = (PROC(REF[]REAL, REF[,]REAL)functions,
REF[]REAL x, REALPAIR accuracy) BOOL

The procedure attempts to place in x the solution of n non-linear equations in n unknowns

$$f_i(x_a, \dots, x_{a+n-1}) = 0 \quad [i \text{ from } 1 \text{ to } n]$$

where a is the lower bound of the array variable x, and could conveniently be taken to be 1. Before newton is called, an estimate of the solution must be placed in x. Newton then obtains new values for the x's, and hands these over to a procedure (functions) which must be written by the user. With these x's as data, this procedure must calculate the n function values and their n^2 partial derivatives with respect to the x's, placing all of these in a 2-D array (the second parameter of "functions"). The parameters of "functions" are arranged as follows:

First parameter - supplied by newton when it calls functions:

x_a	x_{a+1}	...	x_{a+n-1}
-------	-----------	-----	-------------

index from a to a+n-1

Second parameter - computed by functions itself, using above values:

first index from 1 to n	$\frac{\partial f_1}{\partial x_a}$	$\frac{\partial f_1}{\partial x_{a+n-1}}$	f_1

	$\frac{\partial f_n}{\partial x_a}$	$\frac{\partial f_n}{\partial x_{a+n-1}}$	f_n

second index from a to a+n

Newton calls the procedure functions repeatedly until the corrections to the x's are smaller, relatively or absolutely, than specified by the "accuracy" parameter (see Approximation 1). Iteration then ceases, and if the residuals of the equations are a minimum, newton delivers TRUE. If all attempts to improve the approximation lead to increases in the residuals, the value FALSE is delivered, and the "solution" given by the x's should be treated with suspicion (it probably corresponds to an extremum).

SOLUTION OF NON-LINEAR SIMULTANEOUS EQUATIONS WHOSE DERIVATIVES CANNOT BE CALCULATED

```
PROC secsol = (PROC([ ]REAL, REF[ ]REAL)functions  
               REF[ ]REAL x,  
               REAL step,  
               REALPAIR accuracy) BOOL
```

The procedure attempts to solve n equations

$$f_i(x_a, \dots, x_{a+n-1}) = 0 \quad [i \text{ from } 1 \text{ to } n]$$

in the n unknown x's, which are indexed from the lower bound a to the upper bound a+n-1. (The value of a could conveniently be taken to be 1.) The array variable x should be set up to contain a guessed solution before secsol is entered, and in many cases the attainment of a successful solution will depend critically on this initial guess.

Throughout the running of secsol, x will contain the best solution found so far, 'best' being defined as that which minimizes the sum of squares of the f's.

The equations to be solved are defined by a procedure which must be written by the user. The name of this procedure ("functions" say) must be supplied as the first parameter of secsol. The procedure "functions" must be written to take x's from its []REAL parameter (bounds a to a+n-1) and put the corresponding functions in its REF[]REAL parameter (bounds 1 to n).

The REAL parameter "step" is used by secsol as an increment to each of the x's to find approximations of the partial derivatives of the functions. It should be chosen small enough to stay in the neighbourhood of x in which the functions may be considered approximately linear, but not so small that the subtractions

$$f_i(\dots, x_j + \text{step}, \dots) - f_i(\dots, x_j, \dots)$$

produce rounding errors.

The procedure will stop and deliver TRUE when the last improvement to the x's is fractionally or absolutely within the error given in the REALPAIR accuracy (see Approximation 1). The result FALSE will be given if a solution cannot be found. In this case, try starting from a different initial guess, or relaxing the accuracy requirement.

MINIMIZATION OF A SUM OF SQUARES OF FUNCTIONS WHOSE DERIVATIVES CANNOT BE CALCULATED

```
PROC minsumsq = (PROC([ ]REAL, REF[ ]REAL)functions,  
                  REF[ ]REAL x,  
                  INT m,  
                  REAL step, REALPAIR accuracy ) REAL
```

The procedure attempts to minimize

$$\sum_{i=1}^m f_i(x_a, \dots, x_{a+n-1})^2 \quad [m \geq n]$$

with respect to the n unknown x 's, which are indexed from lower bound a (which can conveniently be 1) to upper bound $a+n-1$. The array variable x should be set up to contain a guessed solution before `minsumsq` is entered; this should be as close as possible to the required minimum, as it is easy for the procedure to get stuck in a local minimum. Throughout the running of `minsumsq`, x will contain the best solution found so far.

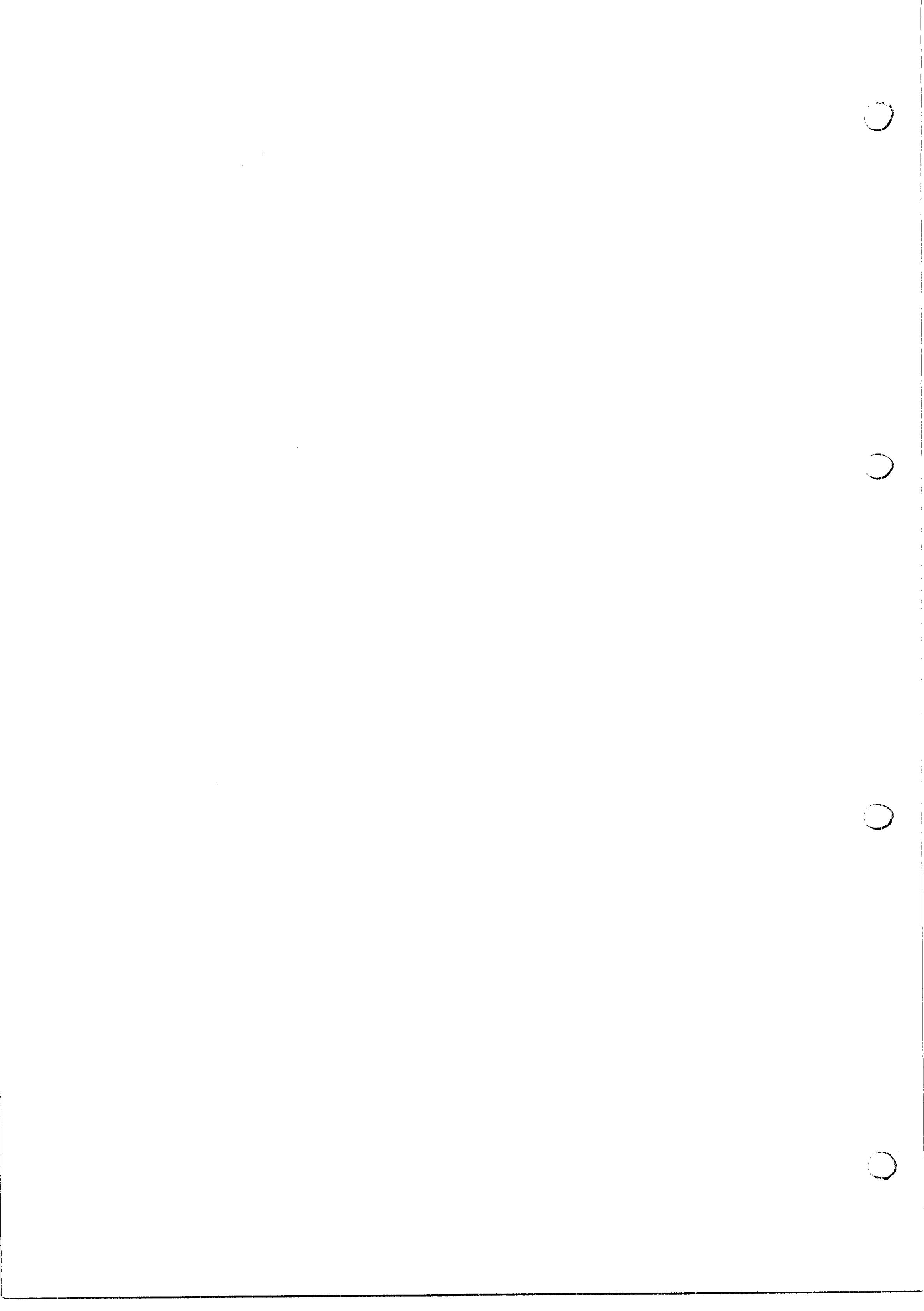
The functions f are defined by a procedure which must be written by the user. The name of this procedure ("functions" say) must be supplied as the first parameter of `minsumsq`. It must be written to take x 's from its `[]REAL` parameter (bounds a to $a+n-1$) and put the corresponding functions in its `REF[]REAL` parameter (bounds 1 to m).

The `REAL` parameter "step" is used by `minsumsq` as an increment to each of the x 's, to find approximations of the partial derivatives of the functions. It should be chosen small enough to stay in the neighbourhood of x in which the functions may be considered approximately linear, but not so small that the subtractions

$$f_i(\dots, x_j + \text{step}, \dots) - f_i(\dots, x_j, \dots)$$

produce rounding errors.

The procedure will stop when the last improvement to the x 's is fractionally or absolutely within the error given in the `REALPAIR` accuracy (see Approximation 1). The result delivered is the final sum of squares of the functions.



1 Introduction

There are many programming situations where output in the form of graphs or pictures will show much more meaning than purely textual output. The system described here provides a way of building up picture definitions, or picture descriptions, within an Algol 68-R program, in a manner that is independent of the output devices or plotters. These pictures can then be drawn very simply with any desired scaling on a choice of plotters.

The main concept used to describe pictures to be drawn is a SKETCH, which signifies a piece of drawing, using the library mode REALPAIR for positions as coordinate pairs. SKETCH is a mode name, and items of this mode are handled in a program in the same ways as items of any other mode. Once provision has been made for the storing of intermediate picture drawing information (see section 1.2), sketches can be manipulated as easily as numbers. In particular, they can be added together to make complex sketches from simpler ones. Also, any sketch can be moved about or repeated by transformations, which are applied using the multiplication symbol. In these ways, a complicated picture can be described as the sum of its major parts, and each major part as the sum of its parts and so on. Another mode called TRAIL provides a way of constructing a sketch in a stepwise manner, and is used when part of a picture must be defined gradually, during the progression of a program. For straightforward graph drawing, the procedures described in sections 5 and 10 can be used to produce complete graphs as sketches. In many programs, such as the example program of section 1.1, a complete picture can be built up by adding together just two or three of these.

The final sketch is fitted onto a rectangular picture area which corresponds to the physical output of a plotter. Unless the sketch is to be drawn with a fixed scaling to fit the output medium in a carefully controlled way, this fitting usually involves specifying the range of coordinate values that are to be visible, or else the sketch itself can be made to fill the picture area automatically, leaving margins suitable for annotation. The fitted sketch has the mode PICTURE, and annotation in a fixed part of the picture area, also of mode PICTURE, can be added to it before it is drawn on a given plotter.

The different types of plotter that can be used are given in section 12. At RRE the microfilm plotter is best for most cases, unless accurate scaling is required. Pictures may also be drawn in a rudimentary form on a line printer; this is particularly recommended for testing new pictures or individual parts of a picture, as it gives the fastest turn-round. Any new plotters that may become available can be added very simply, once a few basic routines to control them have been written.

LIBRARY

Graphical Output

1.1

1.1 An example program

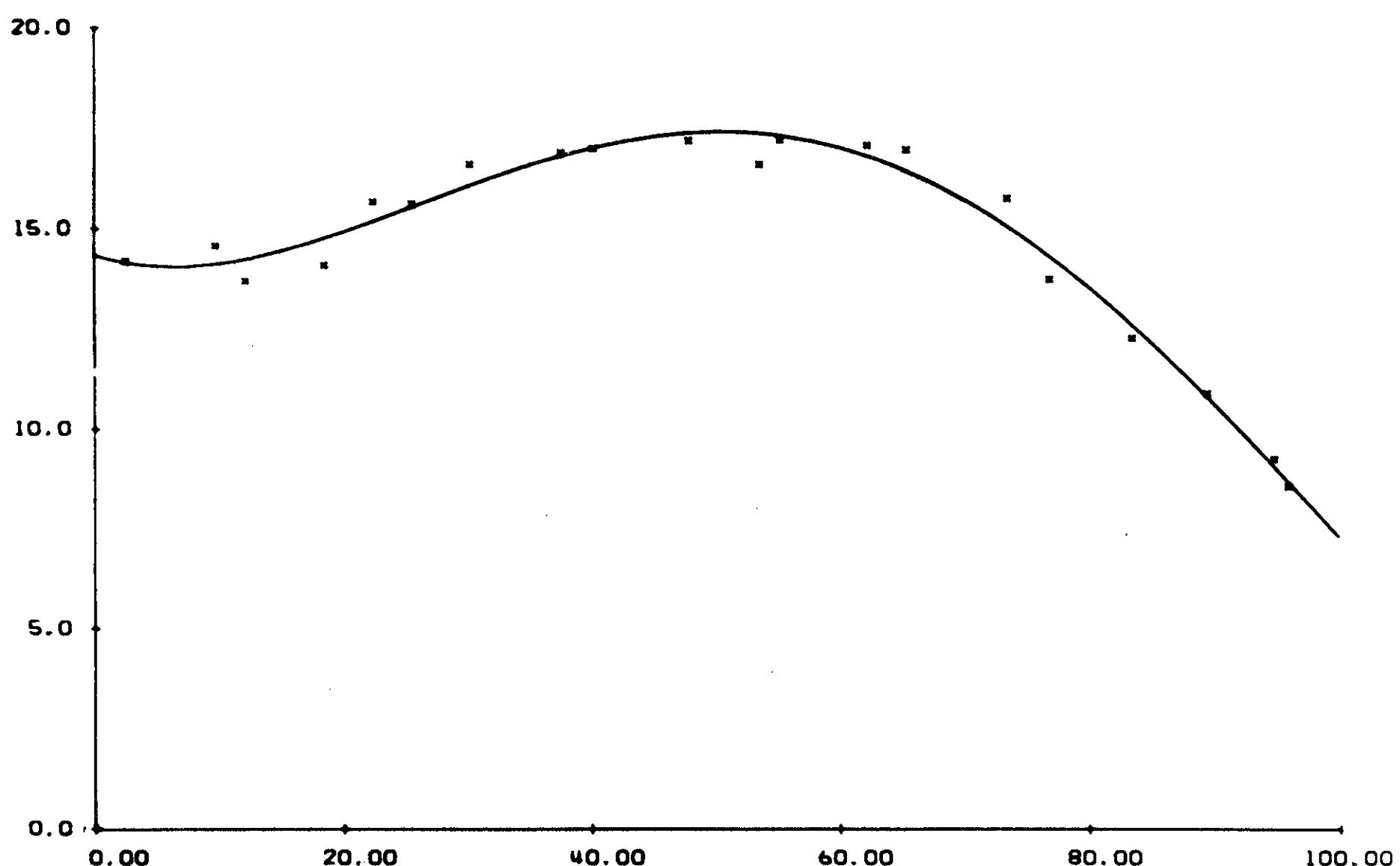
The example program below shows how graphical output may be obtained to illustrate polynomial curve fitting. It first reads in some experimental points, which are in the range (0-100, 0-20), and then calculates a fourth degree polynomial to fit through them, making use of the library procedures "polyfit" and "polynomial". The picture it draws is shown opposite. The line numbers have been added for reference by the notes below.

```
0     example program WITH graph file, pictures, sketch graphs, plot number,
1             micro plotter FROM :devlib.graphslib
2 BEGIN
3     INT channel = 1;  start graph file (channel);
4
5     INT n;  read(n),
6     [1:n] REALPAIR points;  read(points);
7     REAL xmin = 0.0, xmax = 100.0, ymin = 0.0, ymax = 20.0;
8
9     SKETCH given points = sketch points (points, unjoined, cross mark);
10
11    [0:4] REAL coefficients;  polyfit (points, coefficients);
12    PROC curve function = (REAL x) REAL :
13        BEGIN polynomial (x, coefficients) END;
14
15    SKETCH curve = sketch function (curve function, xmin '<>' xmax, 5.0);
16    SKETCH my axes = axes ((xmin, ymin), 20.0, 5, $<3.2>$,
17                           5.0, 4, $<2.1>$);
18
19    PICTURE picture = fit (given points + curve + my axes,
20                            fit range (xmin '<>' xmax, ymin '<>' ymax) )
21                            + picture heading ("POLYNOMIAL CURVE FITTING");
22
23    draw picture (micro plotter, picture);
24
25    end graphs
26 END
27 FINISH
```

Notes

- 0,1 The graphical output routines have to be incorporated in this manner.
See section 1.2.
- 3 Intermediate picture information is stored in a binary file assigned to channel number 1. See sections 1.2 and 11.
- 9 This sketch consists of small crosses to mark the experimental points.
See section 5.2.
- 11 The polynomial coefficients are calculated.
- 12,13 "curve function" is the polynomial in the form of a function.

POLYNOMIAL CURVE FITTING



- 15 This is a sketch of the polynomial over the range $x_{\min} - x_{\max}$. The value 5.0 determines how accurately the curve is to be drawn. See section 5.3.
- 16,17 This sketch consists of a pair of axes. The points to be marked and the number formats to be used are specified. See section 10.
- 19,20 The sum of the three sketches (which is itself a sketch) is fitted to the picture area, by specifying the range of coordinate values required. This produces a PICTURE. See section 8.
- 21 The picture also includes a heading. See section 8.2.
- 23 The picture is to be drawn by the microfilm plotter, on an aperture card. See sections 8 and 12.
- 24 The procedure "end graphs" must always be obeyed when the graphical output from a program has finished. See sections 1.2 and 11.3.

1.2 Obtaining graphical output

The program segments containing the graphical output routines are not held in the system library. They are in an album called

```
:devlib.graphslib 1
```

and the list of segment names required must be given at the head of any program or segment using them, as described in the UTILITIES Albums section of the Programmers Manual. The order of the segment names does not matter. If a private album is being used, it must be initialized to :devlib.graphslib¹, and the graphical output segment names must precede the names of segments from the private album.

The segment names required are those containing the graphical output items used by a program or segment. The appropriate names will be found with the specifications that follow, and also in the Graphical Output Index following section 12. In summary, the usual segments required are:

pictures	Always.
sketches	Usually.
sketch graphs	If any of the graph procedures of section 5 are used.
plot number	If any numbers are to be drawn.
graph file	If picture information is to be held on backing store.
lp plotter	
pen plotter	
micro plotter	If pictures are to be drawn by the appropriate plotters ² . Any number may be used together.
....	

If "sketch graphs" or "plot number" is incorporated in any program, "sketches" will be also, but it must be given explicitly if any of the items declared in it are to be used.

The description or building up of sketches and pictures involves storing intermediate picture information, to be used when the pictures are actually drawn. A program producing graphical output must always start by calling a procedure to indicate where this information is to be stored. The options for this, and how to calculate the amount of picture information that will be generated, are described in section 11. Most commonly, a direct access or binary file is used to hold the picture information, and this is obligatory if an off-lined plotter is used. The procedure to use in this case is "start graph file (channel number)", or "add to graph file (channel number)" if the file already contains picture information that must not be overwritten.

The microfilm plotter and the pen plotter are off-lined. That is, they are not attached directly to a program wishing to output to them, so there is no need to hold up jobs when they are not available. Whenever a picture is requested to be drawn on an off-lined plotter, the information is held up until the procedure "end graphs" is called, which must be called at the end of any program with graphical output. The necessary arrangements are then made for drawing all the off-lined pictures, at the convenience of the computer operators. Naturally, it is important that the file used for storing the picture information is not erased or overwritten before all the pictures have been drawn successfully.

¹ This name applies at RRE. Please check the local installation name.

² The types of plotters available depend on the local installation.

2 Some modes and operators

The following are all declared in the segment "pictures" except where indicated otherwise.

RANGE	This is the mode STRUCT(REAL min, max) which is used by some of the graphical output procedures, though it could have wider application.
'<>'	This is a priority 5 operator (binding less tightly than +), used between any combination of REAL and INT to provide a RANGE. The range min '<>' max could as easily be created by the collateral construction (min, max), but the former is recommended to avoid confusion with the mode REALPAIR.
SKETCH	This is a mode which denotes a piece of picture definition whose internal positions are given by the mode REALPAIR, and which can be altered by transformations.
+ PLUS	Two SKETCHes added together produce a SKETCH which is the straightforward combination of the two. PLUS has its usual meaning - if s1 is a sketch variable and s2 is a sketch, s1 PLUS s2 means s1 := s1 + s2 .
blank sketch	This is the name of the null sketch, which can be added to another sketch with no effect. It can be used to initialize a sketch variable before adding into it with PLUS.
PICTURE	This mode denotes a piece of picture definition which is fixed in a rectangular area corresponding to a physical plotter output, and is not subject to any more transformations. A SKETCH becomes a PICTURE by fitting it to this area, as described in section 8.
+ PLUS	Two PICTURES added together produce a PICTURE which is the combination of the two, on the same picture area. PLUS has its usual meaning.
blank picture	This is the name of the null PICTURE which can be added to another PICTURE with no effect.
FITTING	This mode is used to give the manner in which a SKETCH is to be fitted to a picture area. See section 8.
PLOTTER	This is the mode used for graphical output devices.
TRANSFORMATION * ↑	This is the mode of a transformation that can be applied to a sketch to move it. The operator * between a TRANSFORMATION and a SKETCH produces the transformed SKETCH, and * between two TRANSFORMATIONS or ↑ between a TRANSFORMATION and an INTEGER produce compound TRANSFORMATIONS. See section 6.
SKETCHMOD /	This mode is used for a modification to a sketch which does not alter its position, but affects the way the lines within it are drawn. The operator / between a SKETCH and a SKETCHMOD produces the modified SKETCH. See section 7.
TRAIL	This mode and its associated operators are declared in the segment "sketches" and described in section 4. Trails are used for constructing sketches as sequences of steps.

3 Primitive sketches

The following are declared in the segment "sketches":

```
PROC line = (REALPAIR a, b) SKETCH
PROC rectangle = (RANGE x range, y range) SKETCH
PROC circle = (REALPAIR centre, REAL radius) SKETCH
PROC arc = (REALPAIR start, centre, REAL angle) SKETCH
PROC plot string = ([]CHAR character string, REALPAIR position) SKETCH
```

They are defined as follows:

line (a, b)

This is the straight line between the two positions a and b.

rectangle (left '<>' right, bottom '<>' top)

This consists of four straight lines with x running from left to right and y from bottom to top, and is better than adding together four separate line sketches. Note that other shapes consisting of a sequence of straight lines joined end to end can be produced by a trail, described in section 4.

circle (centre, radius)

This is the circle with given centre and radius. Note that it is circular with respect to its own scaling units, and will not remain circular under all transformations or methods of fitting to a picture area.

arc (start, centre, angle)

This is an arc of a circle with the given centre. The position "start" is one end of the arc, which extends "angle" radians anti-clockwise round the centre. The value "angle" can be negative if the arc is to be drawn clockwise.

plot string (character string, position)

This sketch consists of the single position given, to be labelled with the character string. Only the position will be affected by any transformations, and the character string will be printed horizontally, using the standard character size for the plotter being used, with its first character centred at the given position. Care should be taken when considering how to fit the sketch to the picture area to make sure that there is room to print the whole string. The fitting "fit sketch range" will not take this into account. Other facilities for output of text and numbers are given in section 9.

Examples

```
SKETCH window = rectangle (0 '<>' 2, 0 '<>' 2)
                  + line ((0, 1), (2, 1)) + line ((1, 0), (1, 2));

SKETCH bubbles := blank sketch;
TO 20 DO bubbles PLUS circle ((random, random), random/4);
```

3.1 Point plotting symbols

The identifiers in the table below are all declared in the segment "pictures".

dot mark	.	1
plus mark	+	2
cross mark	×	3
circle mark	○	4
square mark	□	5
diamond mark	◇	6
dot in circle	◎	7
dot in square	▣	8
dot in diamond	◆	9
plus in circle	⊕	10
cross in circle	⊗	11
plus in square	■	12
cross in square	▣	13
plus in diamond	◆	14
cross in diamond	❖	15
asterisk mark	*	16

Each of these is a SKETCH, and consists of the point plotting symbol shown alongside in the table, to be drawn at the position (0, 0), though they are normally used in conjunction with some procedure which provides a position, such as "plot" described below.

```
PROC plot = (SKETCH symbol, REALPAIR position) SKETCH
```

This procedure is declared in the segment "sketches". If it is used with one of the point plotting symbols given above, it will produce a sketch consisting of the symbol to be drawn at the given position. In general, the procedure transforms any sketch by shifting it so that the origin moves to the given position.

The following procedure is declared in the segment "pictures":

```
PROC symbol = (INT n) SKETCH
```

This is used when a particular point plotting symbol is to be computed. Given a number in the table above, it produces the appropriate symbol, so that "symbol(1)" is equivalent to "dot mark" etc. If the number given is not in the table above, the symbol drawn is unspecified and will depend on which plotter is used.

The point plotting symbols are used merely to mark positions, and are otherwise unaffected by transformations, the centre of a symbol always being at the required position. If any particular plotter can draw different symbols more easily than those given above, they will be drawn instead, as specified in section 12, but the identifiers in the table above should always be used.

4 Trails

Sketches consisting of a sequence of lines joined end to end, such as graphs, can be constructed in a sequence of steps by means of a TRAIL. A trail is started at a given position by "start trail", and extended by trail movements (such as "draw to") using the operator ' \rightarrow ', and a sketch is formed from it by "sketch trail". This is more convenient than adding together separate line sketches as the end points do not have to be repeated, and if broken or dotted lines are being used the dot pattern is maintained over all the lines in a trail.

The following are all declared in the segment "sketches".

TRAIL The mode used for making up a sketch in a sequence of steps.
A trail consists of a path from some starting point to a current position.

PROC start trail = (REALPAIR p) TRAIL
This is used to start a new trail at a position p.

TRAILMOVE This mode denotes a trail movement, used to extend a trail.
' \rightarrow ' This is a priority 5 operator used between a TRAIL and a TRAILMOVE, in that order, to produce a TRAIL which is the old one extended according to the TRAILMOVE.
' \rightarrow :=:' This is a priority 1 operator used between a REF TRAIL (or TRAIL variable) and a TRAILMOVE, in that order, which does the same as ' \rightarrow ' with assignment to the REF TRAIL, so that t ' \rightarrow =' m means t := t ' \rightarrow ' m .

PROC draw to = (REALPAIR p) TRAILMOVE
This trail movement will extend a trail with a straight line from its current position to p, the new current position.

PROC move to = (REALPAIR p) TRAILMOVE
This trail movement will merely break off a trail and start it again at p, like "draw to" but with no line drawn.

REL This is a monadic operator which is applied to a TRAILMOVE if the position in the trail movement is given relative to the current position of the trail being extended.

PROC sketch trail = (TRAIL t) SKETCH
This is used to obtain the required sketch from a trail.

Examples

```
SKETCH capital m = sketch trail ( start trail ((0, 0)) ' $\rightarrow$ '  
                          draw to ((0, 1)) ' $\rightarrow$ ' draw to ((0.5, 0.5)) ' $\rightarrow$ '  
                          draw to ((1, 1)) ' $\rightarrow$ ' draw to ((1, 0)) );
```

The following will create the same sketch, illustrating ' \rightarrow =' and REL.

```
TRAIL m trail := start trail ((0, 0));  
m trail ' $\rightarrow$ =' REL draw to ((0, 1));  
m trail ' $\rightarrow$ =' REL draw to ((0.5, -0.5));  
m trail ' $\rightarrow$ =' REL draw to ((0.5, 0.5));  
m trail ' $\rightarrow$ =' REL draw to ((0, -1));  
SKETCH another m = sketch trail (m trail);
```

4.1 Drawing curves

Trails can also be used for drawing curves. The following are declared in the segment "sketches".

```
PROC curve draw to = (REALPAIR p) TRAILMOVE
PROC function draw to = (REALPAIR p) TRAILMOVE
PROC curve move to = (REALPAIR p) TRAILMOVE
PROC function move to = (REALPAIR p) TRAILMOVE
PROC tangent through = (REALPAIR p) TRAILMOVE
```

A curved trail movement is obtained by using "curve draw to" in place of "draw to", and successive curved movements in a trail will form a continuous smooth curve through the individual start and end points. The way the curve is drawn is described in section 4.1.1, and the possibility of inaccuracies described there should be noted. If the successive x values on a curve always increase or always decrease, so that the curve can represent y as a function of x, a better curve is usually obtained by using "function draw to" instead of the more general "curve draw to".

The way the directions of the two ends of a curve are calculated is also described in section 4.1.1. These will be at the ends of a trail, or between different types of trail movement, including between the two types of curved movement. If a more definite direction is required, the curve can be projected beyond its ends by non-drawing curved movements using "curve move to" or "function move to", which otherwise behave as "move to", as shown in the second example below. Alternatively, a tangent direction can be given by using the special trail movement "tangent through (p)" giving any point "p" on the required tangent in the forward direction, possibly using the operator REL to give just the direction of the tangent. The current position of the trail is not altered in this case.

Examples

If a number of positions are held in an array [1:n]REALPAIR points, a function type curve through them, assuming the x values are monotonic, would be obtained by

```
TRAIL curve := start trail (points[1]);
FOR i FROM 2 TO n DO curve '->:=>' function draw to (points[i]);
sketch trail (curve)
```

If the curve were closed, with points[n] joined to points[1], the general type of curve would be used, projected at either end to ensure smoothness at the join, as follows:

```
TRAIL curve := start trail (points[n-1]) '->' curve move to (points[n]);
FOR i TO n DO curve '->:=>' curve draw to (points[i]),
sketch trail (curve '->' curve move to (points[1]))
```

Note that each of these examples could be replaced by a call of the procedure "plot points", which is described in section 5.2.

4.1.1 Inaccuracies in curve drawing

A curve through a number of fixed points is drawn by calculating separate sections of curve for each pair of adjacent points, such that the direction at each point is continuous. The accuracy of the resulting curve depends on the frequency of the given points, taking account of the curve calculation method described below. If they are too far apart, the resulting curve may take some unexpected turns. In particular, it is usually quite unsuitable to draw a curve through a set of points which are subject to experimental error; such points should be joined by straight line segments, or else some curve fitting method should be used to provide an approximating curve.

Two methods of calculating curves are used. The first (more general) method is used when nothing special is assumed about the coordinate directions, whereas the second (function type) method is used when the curve represents y as a function of x , with successive values of x increasing or decreasing monotonically. With the first method, all transformations are applied to the given points before calculating the curve sections, which use the final positions directly. Simple rotation or uniform expansion will not change the appearance of such a curve, but independent scaling of x and y can. With the second method, the curve is calculated using the coordinates as given, and all transformations are applied to the whole curve.

Each section of the first type of curve is a vector cubic with both direction and curvature (approximately) at each given point being that of the circle through the point and its two neighbours. With the second type of curve, each section is a simple cubic with the tangent at each given point being that of the quadratic or parabola through the point and its two neighbours. At the two ends of both types of curve, if the direction is not given explicitly, it is chosen to make the curve converge to a straight line at the end points. Apart from the end sections, the first type of curve will be a very close approximation to a circle, when given some points on one, whereas the second type of curve will follow a quadratic exactly.

In most applications, the curve is required only to look smooth, and the frequency of given points to achieve this can be found out by trial. In those cases where accuracy is more important, the following two procedures may be used to monitor it. They are declared in the segment "sketch graphs".

```
PROC max curve error = (REALPAIR a, b, c, d) REAL
PROC max function error = (REALPAIR a, b, c, d) REAL
```

The value "max curve error (a, b, c, d)" is the maximum distance between the arcs of circles a-b-d and a-c-d, plus the error of approximating a circle by the first curve drawing method. If a curve is drawn through four points a-b-d-e by the first method, the section b-d will lie between the arcs a-b-d and b-d-e, and so the distance to the drawn curve from a point c, which ought to lie on the true curve between b and d, cannot be greater than either of "max curve error (a, b, c, d)" or "max curve error (b, c, d, e)". The value given by "max curve error" will be in the units of the given points, which will have no relevance if unequal scaling is applied. When the second method is used, the corresponding value is "max function error (a, b, c, d)", which gives the maximum y value between a and d between the two quadratics through a-b-d and a-c-d respectively.

4.2 Symbols and other extensions to trails

These are all declared in the segment "sketches".

The operator ' \rightarrow ' (and similarly ' $\rightarrow:=$ ') described in section 4 can also be used between a TRAIL and a SKETCH to produce a TRAIL, which is the old one extended to include the SKETCH which is moved so that its origin lies at the current position of the TRAIL. The current position of the extended trail is unchanged. For example, if "t" is a TRAIL variable, the point plotting symbol "cross mark" would be included at its current position by

```
t ' $\rightarrow:=$ ' cross mark
```

A sketch drawn in the middle of a curve in this way will not affect the way the curve is drawn.

An arc of a circle can also be drawn as part of a trail by using

```
PROC trail arc = (REALPAIR centre, REAL angle) TRAILMOVE
```

The trail movement "trail arc (centre, angle)" will extend a trail by an arc of a circle with the given centre, starting from the current position of the trail, and extending "angle" radians anti-clockwise round the centre to what will be the current position of the extended trail. The centre can be given relative to the old trail position if the operator REL is used.

5 Drawing graphs

The procedures described in this section make use of the primitive facilities described in previous sections, and have been written to simplify some commonly occurring graph drawing requirements.

5.1 A graph of values known at equally spaced argument values

```
PROC sketch graph = ([]REAL y array, RANGE x range,  
INT join, SKETCH symbol) SKETCH
```

This procedure is declared in the segment "sketch graphs", and is used to produce a graph as a sketch, where a given set of y values in "y array" corresponds to equally spaced values of x, which are given as a range in the form "first x '<>' last x". The parameter "join" defines how the points on the graph are to be joined and should be one of the following identifiers, which are also declared in the segment "sketch graphs".

- unjoined
- straight joins
- curved
- function join

If this parameter is "curved" (or "function join" which is exactly equivalent here), a smooth curve is drawn through all the points using the second (function type) method described in section 4.1.1, and the warning there on inaccuracy should be noted.

Whether the points are joined or not, symbols can also be drawn at each point. The sketch given for the parameter "symbol" will be drawn with its origin at each of the points. This parameter is most suitably one of the point plotting symbols listed in section 3.1, or "blank sketch" if symbols are not required.

Example

If a REAL array y with bounds 0 : 10 contains a set of values, with $y[i]$ corresponding to $x = i/10$, the following sketch would be used to draw a graph using straight line segments without point plotting symbols.

```
sketch graph (y, 0 '<>' 1.0, straight joins, blank sketch)
```

5.2 A graph given as a set of positions in an array

```
PROC sketch points = ([]REALPAIR points, INT join, SKETCH symbol) SKETCH
```

This procedure is declared in the segment "sketch graphs", and is used to produce a graph as a sketch, where a set of positions are given in the array "points". The parameter "join" defines how the points on the graph are to be joined and should be one of the following identifiers, which are also declared in the segment "sketch graphs":

- unjoined
- straight joins
- curved
- function join
- joined polygon
- closed curve

In the last two cases, the array is considered cyclic in the sense that the last position in it is adjacent to the first, and "joined polygon" is the same as "straight joins" with the addition of a straight line between the first and last positions. A smooth curve is drawn through all the points using the first (more general) method described in section 4.1.1 if the given parameter is "curved" or "closed curve", or the second (function type) method if it is "function join", which should be used only if the x values increase or decrease through the array. Note the warning on inaccuracies of curve drawing given in section 4.1.1.

Whether the points are joined or not, symbols can also be drawn at each point. The sketch given for the parameter "symbol" will be drawn with its origin at each of the points. This parameter is most suitably one of the point plotting symbols listed in section 3.1, or "blank sketch" if symbols are not required.

Example

This example, taken from the program of section 1.1, uses the procedure for unjoined points marked with small crosses.

```
sketch points (points, unjoined, cross mark)
```

5.3 Drawing an analytical smooth curve

PROC sketch function =

(UNION(PROC(REAL)REAL, PROC(REAL)REALPAIR) function,
RANGE range, []REAL step) SKETCH

This procedure is declared in the segment "sketch graphs", and is used to produce a sketch for drawing the smooth curve defined by the procedure "function". This function may either give values for y in terms of x (PROC(REAL)REAL), or else give both x and y in terms of some other parameter (PROC(REAL)REALPAIR). The part of the function required is given by specifying the parameter range, in the form "start '<>' finish".

The function is evaluated only at a fixed number of parameter values to obtain a set of points on the curve, which will be joined in the manner described in section 4.1.1, using the first method in the case of a PROC(REAL)REALPAIR, or the second in the case of a PROC(REAL)REAL. The points obtained depend on the parameter "step" as described below. They will not be specially marked unless suitable action is taken within the "function" procedure.

The parameter "step" is usually given as a single REAL number, in which case the function is evaluated at regular parameter intervals of this amount, which must be small enough for the points to define the shape of the curve clearly. Making it smaller will make the curve be drawn more accurately, but at the cost of more function evaluations, more picture space used (5 words for each point obtained), and more time taken to draw it. The function must also be capable of evaluation with the parameter up to this amount outside the given range, in order to determine the direction at each end of the curve.

If it is necessary to obtain a specified accuracy, the parameter "step" can be given as a pair of REAL numbers. The first of these is used as a parameter interval as described above to obtain an initial set of points on the curve. The second number given is the accuracy required. This is the maximum error of y allowed in the case of a PROC(REAL)REAL, or absolute distance in the case of a PROC(REAL)REALPAIR, which will be meaningful only if equal scaling is applied to the x and y coordinates. Extra intermediate points on the curve are calculated as necessary to ensure that if each point individually were not used, it would be within the given accuracy from the curve defined by the remaining points (see section 4.1.1). If the first number is chosen sensibly, the whole curve should be well within the given accuracy.

Examples

The example program of section 1.1 uses "sketch function", calculating the curve at intervals of 5.0. The following two examples each draw one cycle of a sin curve, the first using intervals of pi/12, and the second demanding an accuracy of 0.001 in y.

```
sketch function (sin, 0 '<>' twopi, pi/12)
sketch function (sin, 0 '<>' twopi, (pi/12, 0.001))
```

The next example uses a PROC(REAL)REALPAIR for a figure eight, using 24 points.

```
PROC figure eight = (REAL t) REALPAIR : BEGIN (sin(2*t), 2*sin(t)) END;
sketch function (figure eight, 0 '<>' twopi, pi/12)
```

6 Transformations

The mode TRANSFORMATION and its associated operators * and \dagger are declared in the segment "pictures". A TRANSFORMATION is applied to a SKETCH by the operator * put between them, with the transformation written first, to produce the transformed SKETCH. The transforming of individual points does not take place until the sketch is drawn, so the sketch information is not repeated.

Transformations can also be multiplied together to make compound ones, so that if t_1 and t_2 are transformations and s is a sketch, $(t_1 * t_2) * s$ is equivalent to $t_1 * (t_2 * s)$ and the brackets are not necessary. Note that t_2 is applied to the sketch first. Transformations may similarly be raised to integer powers by the operator \dagger . It is more economical to transform a sketch once by a compound transformation than to transform it successively by the primitive transformations.

Primitive transformations are obtained from the following procedures which are declared in the segment "sketches".

```

PROC shift   = (REALPAIR p) TRANSFORMATION
PROC expand  = (REAL e) TRANSFORMATION
PROC rotate  = (REAL angle) TRANSFORMATION
PROC stretch = (REAL e1, e2) TRANSFORMATION
PROC shear   = (REAL s) TRANSFORMATION

```

shift (p)	moves a sketch through the REALPAIR vector p, taking any position (x,y) to (x + x OF p, y + y OF p).
expand (e)	expands (or contracts) uniformly about the origin by the amount e, taking (x,y) to (e*x, e*y).
rotate (angle)	rotates "angle" radians anti-clockwise about the origin, taking (x,y) to (c*x - s*y, s*x + c*y), where c = cos(angle) and s = sin(angle).
stretch (e1, e2)	expands x and y independently, taking (x,y) to (e1*x, e2*y). For example, "stretch(-1, 1)" gives a mirror image.
shear (s)	shears in the x direction, taking (x,y) to (x + s*y, y).

```
PROC repeated sketch = (SKETCH s, TRANSFORMATION t, INT n) SKETCH
```

The above procedure, which is declared in the segment "pictures", is used to draw a sketch s repeatedly, in n different places. The resulting sketch is s itself with copies obtained by repeating the transformation t on it. "repeated sketch (s, t, n)" is equivalent to " $s + t*s + \dots + t^{\dagger(n-1)}*s$ ".

Examples

```

SKETCH ellipse = stretch(a, b) * circle((0,0), 1);

SKETCH hundred squares =
    repeated sketch (line((0,0), (0,10)), shift((1,0)), 11)
    + repeated sketch (line((0,0), (10,0)), shift((0,1)), 11);

```

6.1 Compound transformations

If a given sketch is to be drawn at a particular position, it may be more convenient to specify the transformation required in terms of the coordinates of points in the sketch, rather than work out the primitive transformations required. The following procedures are used to provide transformations in this form. They are declared in the segment "mappings".

```
PROC map vector = (POSITIONPAIR p, REAL e, angle) TRANSFORMATION
PROC map line   = (POSITIONPAIR p1, p2) TRANSFORMATION
PROC map triangle = (POSITIONPAIR p1, p2, p3) TRANSFORMATION
```

where POSITIONPAIR here means STRUCT(REALPAIR old, new).

They are used as follows:

```
map vector ((old, new), e, angle)
```

This transformation will transform a sketch so that the position with "old" coordinates will appear at the position "new", and there will be an expansion "e" and a rotation "angle" about this point. It is exactly equivalent to "shift(new) * rotate(angle) * expand(e) * shift((-x0Fold, -y0Fold))".

```
map line ((old1, new1), (old2, new2))
```

This will transform a sketch so that the old coordinate positions "old1" and "old2" will appear at "new1" and "new2" respectively. Translation, expansion and rotation may be included in this transformation, but not shearing.

```
map triangle ((old1, new1), (old2, new2), (old3, new3))
```

This will transform a sketch so that the three "old" positions appear at the corresponding "new" positions, provided that they do not lie on a straight line. In this case, all the primitive transformations may be included.

6.2 Logarithmic scales

The use of logarithmic scales can be considered as a different sort of transformation, which does not use the mode TRANSFORMATION. Instead, one of the following procedures is called with a sketch as its parameter. This delivers the sketch with each x or y coordinate, or both, replaced by its natural logarithm. The procedures are declared in the segment "mappings".

```
PROC x log scale = (SKETCH sketch) SKETCH
PROC y log scale = (SKETCH sketch) SKETCH
PROC xy log scale = (SKETCH sketch) SKETCH
```

It should be noted that a straight line between two points will still be drawn as a straight line, between the two transformed points.

7 Different types of line drawing

The mode SKETCHMOD is used for a sketch modification which alters the way the lines of a sketch are drawn. This is done by the operator / put between a SKETCH and a SKETCHMOD, with the modification written last, to produce the modified SKETCH. The mode SKETCHMOD and its associated operator / are declared in the segment "pictures".

If a number of sketches require the same modification, it is more economical to modify their sum once, than to modify them individually. If part of a sketch is already subject to an incompatible modification, this earlier one will take precedence for that part.

7.1 Broken lines

The following SKETCHMODs are declared in the segment "pictures".

broken line
dotted line
chain line
unbroken line

A "broken line" has equally spaced dashes, a "dotted line" has shorter dashes, or dots, and a "chain line" has alternate dashes and dots. The actual size of the dashes, dots and spaces will depend only on the plotter being used, not on any transformations applied to a sketch. The modification "unbroken line" is rarely necessary, as lines are unbroken by default, but it can be used to fix part of a sketch which might be modified later with broken lines.

More varied dash-dot patterns for sketch modifications can be obtained from the following procedures, also declared in the segment "pictures".

```
PROC broken line size = (INT dash, space) SKETCHMOD
PROC chain line size = (INT dash, space, dot) SKETCHMOD
```

The sizes of the dashes and spaces, or dashes, spaces and dots, are given as integer multiples of some dot unit, which depends only on the plotter being used, and is usually about one fifth of the standard character spacing. The modification "broken line size (dash, space)" is exactly the same as "chain line size (dash, space, dash)". If a plotter is able to draw some dot patterns easier than others by special hardware, these will be used for the three standard patterns, otherwise the following correspondence will hold.

```
broken line    =    broken line size (3, 1)
dotted line    =    broken line size (1, 3)
chain line     =    chain line size (3, 2, 1)
```

Example

```
SKETCH dotted sin curve =
    sketch function (sin, 0 '<>' twopi, pi/12) / dotted line;
```

7.2 Other line types

Some plotters have special hardware for drawing horizontal and vertical lines that is more efficient to use than ordinary lines, but might give a different appearance. When available, this is always used by the axis drawing procedures for axes drawn horizontally or vertically. Any sketch modified by the SKETCHMOD

axis line

which is declared in the segment "pictures", will have all its lines drawn unbroken, and any horizontal or vertical lines in it will be drawn by this special hardware if it is available.

Other modifications to cater for different line thicknesses or intensities, or different colours, may be available on some plotters. These will be described in section 12 with the appropriate plotters. Any of these modifications, which are significant only with particular plotters, can still be used with any other plotters, but without having any effect.

The operator / may be used between two SKETCHMODs to produce the combined SKETCHMOD, assuming that the two are compatible, but this is usually less economical than applying the two modifications to a sketch directly, one after the other. It may be useful for combining different plotter-dependent modifications.

8 Drawing pictures

When a sketch is ready to be drawn it must first be fitted to a rectangular picture area, which will fix the scaling. The picture area corresponds to the physical output area provided by each plotter for drawing pictures, and its size can be found in the relevant part of section 12. The following two procedures are used first to do the fitting, which creates a PICTURE, and then to draw the PICTURE:

```
PROC fit = (SKETCH sketch, FITTING fitting) PICTURE
PROC draw picture = (REF PLOTTER plotter, PICTURE picture)
```

They are declared in the segment "pictures", and can be combined in the form

```
draw picture (plotter, fit (sketch, fitting) )
```

The alternative methods of fitting the sketch, using the mode FITTING, are described in section 8.1. The parameter "plotter" will be the appropriate PLOTTER variable given in section 12.

When making up any picture to be drawn on a particular plotter, consideration should be given to any drawbacks of that plotter, as indicated in section 12. Note also that in the case of offlined plotters, "draw picture" serves merely to take note of the pictures to be drawn. This information is held up until the procedure "end graphs" is called (see section 11.3), when the arrangements for drawing them are made.

On some plotters, a picture or frame number is printed alongside each picture, starting with the number 1. This number sequence may be altered if the pictures are to be considered as part of a sequence not all drawn by the same program run, by assigning the first number required to the integer field

```
frame number OF plotter
```

where "plotter" is the appropriate PLOTTER variable. The number sequence can be stopped altogether by the assignment

```
frame number OF plotter := 0
```

LIBRARY

Graphical Output

8.1

8.1 Sketch fitting

The mode FITTING is used to describe how a sketch is fitted to the picture area, using the procedure "fit". Normally, margins round the edge of the picture area are reserved for annotation and the sketch is fitted into the rectangle between the margins. The width of the side margins is $10\frac{1}{2}$ standard character spacings, with $4\frac{1}{2}$ line spacings for the top and bottom. The sketch is allowed to extend into the margins, and any part outside the whole picture area is simply ignored.

The following are all declared in the segment "pictures":

```
PROC fit range = (RANGE x range, y range) FITTING
FITTING fit sketch range
FITTING natural scale
```

The fitting "fit range (xmin '<>' xmax, ymin '<>' ymax)" is used to specify the required range of coordinate values of the sketch. The given minimum and maximum x and y coordinates will be fitted to the picture area margins. Alternatively, the fitting "fit sketch range" may be used to fit whatever the existing bounds of the sketch are to the picture area margins. This involves a (possibly lengthy) search through the sketch to find its bounds, so it should not be used if the bounds are well known. Note that the two coordinate directions will normally be scaled independently. The fitting "fit sketch range" may not work correctly if the scaling for x and y are very different and curves are being drawn by the first (more general) method described in section 4.1.1.

The fitting "natural scale" is used when more control over the scaling is required. In this case, the sketch is assumed to have been defined using the natural units of the appropriate plotter, as defined in section 12, and shifted so that the origin is at the centre. This indicates that no further fitting need be done.

The methods of fitting by "fit range" or "fit sketch range" may be modified by using the following, which are declared in the segment "pictures":

FITMOD	This is the mode used for modifying a method of fitting.
/	The operator / is used between a FITTING and a FITMOD, in that order, to produce the modified FITTING.
whole area	This is a FITMOD that is applied to a FITTING if the range specified is to cover the whole picture area, rather than the area between the margins.
equal scaling	This is a FITMOD that is applied to a FITTING if the two coordinate directions are to have the same scaling, so that shapes such as squares and circles are not distorted.

Example

The largest possible circle would be obtained by

```
fit (circle ((0,0), 1),
      fit range (-1 '<>' 1, -1 '<>' 1) / whole area / equal scaling)
```

8.2 Picture headings and margin axes

It is often desirable to print a heading at the top of a picture, without having to specify a position in the coordinate system of the sketch. This is done by the following procedure, which is declared in the segment "pictures":

```
PROC picture heading = ([]CHAR heading) PICTURE
```

This is used with a character string to produce a PICTURE, which can be added to the PICTURE produced by fitting a sketch. The given character string will be printed centrally in the top margin of the picture area.

The following procedure, also declared in the segment "pictures", can be used in place of "draw picture" to draw a sketch with a picture heading:

```
PROC draw sketch = (REF PLOTTER plotter,  
                    []CHAR heading,  
                    SKETCH sketch,  
                    FITTING fitting)
```

This is just a simplification of

```
draw picture (plotter, picture heading (heading) + fit(sketch, fitting))
```

When a sketch is being fitted to the picture area, it is possible to indicate the scaling by a pair of axes in the bottom and left hand margins, without having to know the corresponding coordinates in the sketch. This is done by

margin axes

which is declared in the segment "pictures" and has the mode FITMARK, for marking a fitting. It is added to the sketch being fitted, with "margin axes" always written last, in the SKETCH parameter position of either "fit" or "draw sketch". Strictly, this parameter may be either a SKETCH or the mode which is the result of SKETCH + FITMARK. The numbers printed will be in the format \$<1.2&2>\$, but if they do not differ in the first three figures, one number will be printed with more figures and the rest printed as differences from this one. For other number styles on axes, the procedures of section 10 should be used.

Example

If the range of y-values were not known in the example program of section 1.1, the following could replace lines 16 to 23, with the numbers printed in a different format.

```
draw sketch (micro plotter,  
            "POLYNOMIAL CURVE FITTING",  
            given points + curve + margin axes,  
            fit sketch range);
```

LIBRARY
Graphical Output
9

9 Text in graphical output

Single strings of characters can be printed at any position in a sketch by using the procedure "plot string" as described in section 3. A common requirement beyond this is to print strings offset from a given position in a sketch, with displacements specified in terms of character and line spacings, without knowing character sizes in terms of the coordinate system of the sketch. This is achieved by the following three procedures. The first two are declared in the segment "pictures" but "plot text" is declared in "sketches".

```
PROC string text = ([]CHAR character string) SKETCH
PROC shift text = (REAL chars, lines) TRANSFORMATION
PROC plot text = (SKETCH text, REALPAIR position) SKETCH
```

The procedure "string text" is used first to create a sketch from a given character string. This can be used like any other sketch, as described in section 9.1, but normally it is used only in conjunction with "shift text" and "plot text" as described here. The transformation "shift text (chars, lines)" can be applied to this sketch, to move it "chars" characters to the right and "lines" lines down, and it can be added to similar sketches for a position labelled by more than one string.

The sketch of text so obtained is made to label a position by the procedure "plot text". The sketch "plot text (text, position)" contains just the one position given, to be labelled by the text parameter. The first character of an unshifted string would be centred at the given position, and the character strings will be printed horizontally, using the standard character size of a plotter. Thus "plot text (string text (string), position)" is the same as "plot string (string, position)". Note that when "fit sketch range" is used, only the position parameter of "plot text" will affect the fitting.

Example

```
plot text (shift text (-2, -1) * string text ("LABEL"),
           position)
```

This would be used to print the string "LABEL" centrally one line above the position it is to label.

9.1 Text as a sketch

The sketch "string text (string)" is a sketch consisting of the characters in the string, with a coordinate system such that the first character is centred at the origin and the characters are one unit apart. The transformation "shift text (chars, lines)" is equivalent to "shift ((chars, - lines * ratio))" where the ratio is that of line spacing to character spacing. When "plot text" is used, its sketch parameter "text" will be drawn upright, with its origin at the given position, and scaled so that one unit in the sketch is equal to the standard character spacing of the plotter being used.

Non-standard character sizes and orientations can therefore be obtained by applying the transformations "expand" and "rotate" to the text sketch produced by "string text" and "shift text" before using "plot text". If the plotter to be used is able to print some characters by special hardware (see section 12) the sizes and orientations of text should normally be limited to what the hardware can produce, as anything else will be drawn as a succession of individual lines and will be far more time consuming.

If some text characters are considered as an integral part of a sketch rather than merely labelling it, "string text" can be used on its own without "plot text". The characters will then be drawn as a succession of individual lines which are treated just like any other lines in a sketch. Note that the sketch produced by "string text" has its origin at the centre of the first character, and the characters are one unit apart. Each character lies within the range $(\pm \frac{1}{3}, \pm \frac{2}{3})$ from its centre, but with the bottom of most characters at $y = -\frac{1}{3}$, and the line spacing is taken to be 2 units if "plot text" is not used.

Conversely, an ordinary sketch may be used in place of the "text" parameter of "plot text" in order to relate its size to the standard character size, thus providing new characters or new point plotting symbols.

When an ordinary sketch is directly combined with some text, it should have character spacing units in both directions. If it is preferable to use character spacing for x units but line spacing for y units, a sketch using this coordinate system can be stretched in the y direction so that the stretched sketch is the same shape using character spacing units in both directions, by using the transformation

stretch text

This is declared in the segment "pictures" and has the mode TRANSFORMATION, and it is equivalent to "stretch (1.0, ratio)" where the ratio is that of line spacing to character spacing, also used by "shift text". This ratio is normally 2.0, but it may be different in some cases where a plotter demands it, such as for line printer pictures. Note that a sketch to be transformed by "stretch text" should have y coordinates as a number of lines upwards, whereas "shift text (chars, lines)" shifts a number of lines downwards.

9.2 Numbers and the use of CHARPUT variables

```
PROC plot number = (UNION(INT, REAL) number, FORMAT format,  
                      REALPAIR position) SKETCH  
PROC number text = (UNION(INT, REAL) number, FORMAT format) SKETCH
```

The above two procedures, declared in the segment "plot number", are used for printing numbers in graphical output. The procedure "plot number" is equivalent to "plot string" and "numbertext" to "string text", with a character string consisting of the given number as output through the given format. The format should contain an INT or REAL pattern, possibly with insertions, to match the INT or REAL number given (see LIBRARY Input/Output 4). Thus the following two sketches are identical:

```
plot number (pi, $<1.3>$, position)  
plot string (" 3.142", position)
```

It is also possible to set up a CHARPUT variable to produce graphical output text by using the facilities described in LIBRARY Input/Output 2.5.5 and 5.3.5, as shown in the example piece of program below.

```
[1 : line width required] CHAR s;  
CHARPUT v;  
openc (v, string printer, 1);  
next string (v, s);  
SKETCH text := blank sketch;  
event OF v := (INT event number) INT :  
    BEGIN IF event number = -1 OR event number = -2  
        THEN      COMMENT end of line COMMENT  
            IF char number (v) > 1  
                THEN  text PLUS  
                    shift text (1, line number (v))  
                    * string text (s [ : char number (v) -1])  
            FI  
        FI;  
        -1  
    END;  
    -- - {output through v using put, outf etc}  
  
closec (v);  
text
```

This example produces text suitable for use with "plot text", with the first character of the page to be printed immediately to the right of and below the position given in "plot text". Note that either "closec" must be called with v as a parameter or output must end with a new line for the last line to be added to "text".

9.3 Other character fonts

PROC case text = ([]CHAR character string, INT font) SKETCH

This procedure, declared in the segment "pictures", is used in place of "string text" to draw characters in the fonts shown below. Font 1 is the standard graphic character set used by "string text", and the table below shows what corresponding characters are drawn for the other font numbers.

Font 1 0123456789: ; <=>? ! " #£%&' ()*+, - . / @ABCDEFGHIJKLMNPQRSTUVWXYZ[\$]↑↖

Font 2 0123456789: ; <=>? ! " ≠ £ % ¼ ' () * + , - . /
_ abcdefghi jkl mnopqrstuvwxyz { | } ^ ←

Font 4 0123456789: ; <=>? ! "Ά΢√~≈()*+, - . + Ταβγδεφγηιθκλμνοπξρστυψωχυζ±▽□○"

9.4 Picture annotation

The procedure "picture heading" described in section 8.2 is used to print a simple heading at the top of the picture area. Text can be printed in other fixed parts of the picture area, irrespective of any sketch fitting, by the following procedure which is declared in the segment "pictures":

PROC picture text = (SKETCH text, REALPAIR position) PICTURE

The "text" parameter is obtained in the same manner as for "plot text", and the "position" parameter specifies the position of the origin of "text" in the picture area. This position is given in units which are $(\pm 1, \pm 1)$ at the margins of the picture area, so the following pairs are equivalent:

```
picture text (text, position)
fit (plot text (text, position), fit range (-1 '<>' +1, -1 '<>' +1))
```

```
picture heading (heading)
picture text (shift text (- (UPB heading - LWB heading)/2, -3)
              * string text (heading), (0, 1))
```

10 Drawing axes

For simple uses, "margin axes" can be used to draw a standard pair of axes in the margins of a picture as described in section 8.2. The procedures below produce axes as sketches, giving more flexibility for both position and style of labelling. They are all declared in the segment "plot number".

```

PROC x axis = (REALPAIR p, NUMBER dx, INT xno, FORMAT x style) SKETCH
PROC y axis = (REALPAIR p, NUMBER dy, INT yno, FORMAT y style) SKETCH
PROC axes = (REALPAIR p, NUMBER dx, INT xno, FORMAT x style,
              NUMBER dy, INT yno, FORMAT y style) SKETCH

PROC x list axis = (REAL y, [NUMBER x list, FORMAT x style) SKETCH
PROC y list axis = (REAL x, [NUMBER y list, FORMAT y style) SKETCH

```

where NUMBER here means UNION (INT, REAL).

The procedure "x axis" produces a regular axis from the position p through xno increments of size dx (which may be positive or negative), to the position ($x \text{ OF } p + xno * dx$, $y \text{ OF } p$). At each of the ($xno + 1$) step points, including both ends, a tick mark is drawn and the value of x is printed in the format given by "x style". The mode of the increment dx must match the format given for printing the numbers, and if it is an INT the x values will be rounded to the nearest integer.

The procedure "y axis" similarly produces an axis from p through yno increments of size dy , to the position ($x \text{ OF } p$, $y \text{ OF } p + yno * dy$). At each of the ($yno + 1$) step points a tick mark is drawn and the value of y is printed in the format given by "y style". The mode of the increment dy must match the format given for printing the numbers, and if it is an INT the y values will be rounded to the nearest integer.

The procedure "axes" simply produces the sum of two sketches from "x axis" and "y axis", both starting at the position p .

The procedure "x list axis" produces an x-axis at the given "y" value through the given list of values "x list", which are marked and printed in the format given by "x style". These values need not be equally spaced, which makes it suitable for logarithmic scaling. The mode of each x value must match the format given.

The procedure "y list axis" similarly produces a y-axis at the given "x" value through the list of y values "y list", which are marked and printed in the format given by "y style". The mode of each y value must match the format.

For all these axis procedures, the format for the sequence of numbers to be printed must be given using INT or REAL patterns as described in LIBRARY Input/Output 4. A single pattern may be given if all the numbers are to be printed in the same style. The character string produced for each number is printed centrally one line below the point it represents on an x-axis (but note that this character string may start with several spaces), or to the

(contd)

left of the point it represents on a y-axis, with the last character centred two positions from it. Alternatively, the numbers can be printed above an x-axis or starting two positions to the right of a y-axis by putting " / label above" or " / label right" after the format in any of these procedures. For this purpose, the following identifiers have mode BOOL and are declared in the segment "plot number" along with this meaning of "/":

```
label above
label below
label right
label left
```

In all cases, each number is printed horizontally using a standard character size, at a fixed position relative to the point it represents, so it is not advisable to rotate labelled axes. It is also necessary to ensure that the numbers labelling an axis are not printed too close together, by choosing suitable step increments and taking note of what scaling will be applied.

Examples

```
axes ((0, 0), 0.5, 5, $<1.1>$ / label above, -1, 7, $<1>$ )
```

This is a sketch consisting of a pair of axes from the origin to $x = 2.5$ to be marked in steps of 0.5, and down to $y = -7$ in steps of 1. The x numbers will be printed at the top with one digit before and after the decimal point, and the y numbers will be to the left as single digit integers.

```
SKETCH year axis = shift ((-1, 0)) *
    x axis ((1, 0), 1, 12, $ 4x c ("JAN", "FEB", ..., "DEC", "") $);
```

This defines a year axis from $x = 0$ to $x = 12$ at $y = 0$, with the names of the months printed in place of numbers by using an integer choice pattern in the format. The format starts with four spaces so that each month will be printed after the point it represents, the final space being the middle character, and nothing will be printed at the end of the axis. As integer choice patterns always start from 1, the axis must start from the position $x = 1$, so it is shifted to compensate for this. Shifting an axis may also be useful if the numbers to be printed differ from one another only after several figures, to enable one number to be printed as zero (and printed in full elsewhere), with the others printed as differences.

```
SKETCH log axis = x list axis (0, (0.01, 0.1, 1.0, 10, 100),
    $ 3( <1.2> ), 2( <3> ) $ );
```

This defines an axis from $x = 0.01$ to $x = 100$ at $y = 0$, marked at powers of 10, suitable for logarithmic scaling by "x log scale" described in section 6.2.

11 Starting and finishing

The description or building up of sketches and pictures involves storing intermediate picture information. Before calling any procedure or operator which may create picture information, it is necessary to indicate where the information is to be stored, either on backing store or in program data space, by calling one of the procedures described in sections 11.1 and 11.2. Failure to do so will cause the fault display GRAPHS NOT STARTED. If any offlined plotters are to be used, the picture information must be held on backing store. It is also necessary to finish any graphical output as described in section 11.3.

11.1 Picture information on backing store

If the picture information is to be stored on backing store, which is the usual case, a direct access or binary file is used. A suitable file must be created by the George command

CREATE filename (*da, BUCKET 1, KWORDS integer)

where integer is the size of the file required in units of 1024 words. It is important that the file belongs to the same George username as the graph jobs which are to use it. The file name can be given a specific generation number, but a language code must not be used to distinguish it. The file is attached to the program using the George command

ASSIGN *da channel number, filename (OVERLAY)

between the COMPILE and EXECUTE commands, where channel number is in the range 0 to 10. A file on a private disc may be used, in which case the appropriate George command should be given instead of ASSIGN.

In the program, all picture information will be directed to the direct access file assigned to the correct channel number by calling either of the following two procedures, which are declared in the segment "graph file":

```
PROC start graph file = (INT channel number)
PROC add to graph file = (INT channel number)
```

To start a new file for graphical output, "start graph file (channel number)" should be used. If however the file already contains some picture information which must not be overwritten, then "add to graph file (channel number)" should be used, to direct the new picture information beyond the end of all existing picture information. Note that when an offlined plotter is used, the picture information must not be overwritten until all the pictures have actually been drawn. In the case of offlined plotters under the control of the computer operators, the file will be partially protected while pictures are waiting to be drawn, and any call of "start graph file" will then be treated as a call of "add to graph file" instead.

11.2 Picture information in program data space

If the picture information is not to be stored on backing store, it can be held in a BITS array declared in the program, provided that the pictures are local to the program and no offlined plotter is used. This is achieved by calling the following procedure, declared in the segment "local graphs":

```
PROC start local graphs = (REF[]BITS local picture space)
```

The BITS array "local picture space" should be declared large enough to hold all the picture information generated, and must remain in scope until the procedure "end graphs" has been called.

11.3 Finishing

When all graphical output from a program has been completed, the program must end by calling the following procedure, declared in the segment "pictures":

```
PROC end graphs = VOID
```

This closes the binary file, if one is being used for picture information, and also makes the arrangements for drawing on any offlined plotters that may have been used. Note that if a program that is producing several pictures to be drawn on an offlined plotter fails before all the pictures are finished, the earlier pictures will only be drawn if "end graphs" is called via the library procedure "postmortem".

11.4 Picture space

The amount of picture information space required can be estimated very approximately by allowing one word for each procedure or operator called in the creation of pictures, plus whatever is needed to store their parameters - two words for each real number for example, and one word for a complete sketch already defined. Five words are required for a trail movement. It is best to allow for a substantial amount of overhead when first calculating the space required, and the actual amount of space used can be found by the following procedure, which is declared in the segment "pictures":

```
PROC picture space = INT
```

This can be called and printed at any point in the program to give the picture space used so far, in words. It can be called at the end, after "end graphs", to give the total amount of space used. If the picture information is held on backing store, the total amount of information in the file will be given.

(contd)

LIBRARY
Graphical Output
11.4 contd

Picture information space used in the description of a sequence of pictures is not recovered between the pictures. Clearly, it must not be recovered if an offlined plotter is being used. This might be an embarrassment in the case of an online plotter used in an interactive manner, so in this case only, picture information space on backing store or in program data space may be recovered by the following procedure, declared in the segment "pictures":

PROC set picture space = (INT amount)

This is called with the amount of picture space to be preserved, which must have been previously ascertained by the procedure "picture space". All the picture information generated since that time will be lost and the space made available for further use. Great care must be taken that no information still to be used is lost in this way.

11.5 Using pre-defined sketches

If a complicated sketch is built up in one program, it is possible to keep the information for use in another program. One file should be used for picture information from both of the programs, with the second starting by calling "add to graph file", though to save repeated runs of the second program from filling up the file the original file could be copied to a new one each time. The information is passed between the programs by means of an integer address into the picture information file, using the following two procedures declared in the segment "pictures":

PROC sketch address = (SKETCH sketch) INT
PROC retrieve sketch = (INT address) SKETCH

The procedure "sketch address" is used in the first program to provide an integer address for the required sketch. In the second program, "retrieve sketch" is called with this integer address to re-create the sketch. It must only be used with integers that have been derived in this way.

12 Graphical output devices

The different graphical output devices are identified by individual PLOTTER variables, used with the procedure "draw picture" described in section 8. Details of the available devices and the PLOTTER identifiers follow from section 12.2. Depending on the PLOTTER variable used, the picture information is either used directly to draw the picture when "draw picture" is obeyed, or else it is held up for drawing later by a special plotting program. In the latter case, the picture information must be held on backing store, and the arrangements are made for drawing on these offlined plotters when the procedure "end graphs" is called.

The pen plotter and microfilm plotter at RRE are offlined in this way to give the computer operators full control over their use, and to allow user programs to proceed when they are not available. When "end graphs" is called and there are pictures to be drawn by these plotters, an entry is made automatically in a special system file which gives the operators all the information they require, including the name of the picture information file used. A message is also displayed on the user's monitoring file giving the number of pictures to be drawn and the name of the plotter. Since the actual plotting is out of the user's control, care should be taken not to erase or overwrite the picture information file too soon; if it is erased too soon by mistake, the computer operators should be informed.

Other graphical output devices that are available more directly to the user may also be offlined in order to reduce program sizes, as the procedures that interpret the picture information for plotting are rather large. Line printer pictures for example may be offlined as an alternative to drawing them directly. In these cases, the appropriate plotting program must be run by a George macro-command as described in section 12.1.

12.1 Offlined plotters

When the procedure "end graphs" is called and there are pictures to be drawn by any plotter for which the user must run a separate plotting program, a pair of messages is displayed on the monitoring file in the form

DISPLAY : n PICTURES FOR plotter AT :-
DISPLAY : FILE filename (gen) / address

where gen is the file generation number and address is an integer to indicate the relevant position within the picture information file. The second message contains all the information required by the plotting program. If a private disc file is used, the second message is displayed in the form

DISPLAY : FILE (cartridge serial number, filename (gen)) / address

(contd)

LIBRARY

Graphical Output

12.1 contd

Each of the offline plotting programs is of a similar form, run by a George macro-command. In each case the second displayed message must be given as embedded data for the plotting program by carefully copying it into the job description on the line following the relevant macro-command, starting with "FILE", though "(gen)" may be omitted if this does not cause ambiguity. For example, the following may be correct for offlined line printer pictures:

```
LPOFFPLOT  
FILE mygraphfile(1) / 1234
```

This input data may optionally be followed on the same line by further information to control the plotting: "MISS m" will make the plotting program skip the first m pictures, and "ONLY n" will make it stop after plotting n pictures, which is useful if it is necessary to replot a small number of pictures from a sequence. It may also be useful to detect errors of scaling by putting "FIT", which merely over-rides any fitting used by the procedure "fit" (see section 8) and substitutes "fit sketch range" with margin axes. It is also possible to draw the pictures from more than one program run at a time, by putting the data for each run on a separate line, each starting with "FILE", and ending each line except the last with the character "&".

An offline plotting program cannot be run in the manner described above in the same job as the program producing the pictures without an interaction by the user to copy the displayed message back in. To avoid this, it is possible to put the message into a character file and arrange for it to be read from there by the plotting program. To do this, the following procedure should be called at any time before "end graphs". It is declared in the segment "offline plotters".

```
PROC offline plot = (REF PLOTTER plotter, PROC([ ]CHAR) procedure)
```

The given parameter "procedure" will be called during "end graphs" if any pictures are to be drawn by the given plotter, and its parameter will be the same character string as the second display message. The procedure should be written so that its parameter is output as the first and only line of some new character file, and it must be still in scope at the call of "end graphs". To make the plotting program read this file instead of embedded data the macro-command is given a parameter of the form

FILE filename

where filename is the name of the file containing the data (which also starts with the word "FILE").

12.2 Line printer pictures

Pictures drawn on a line printer are produced directly from the program, along with other program output. They are drawn in a rather rudimentary manner, with sequences of dots for lines, but this is often perfectly adequate for simple pictures not cluttered by too many lines in any part. It is also quite useful for obtaining a quick check for complicated pictures by drawing them by line printer before using one of the offlined or slower plotters.

The following are declared in the segment "lp plotter":

```
PLOTTER stand lp plotter;  
PROC set lp plotter = (REF PLOTTER pv, REF CHARPUT cv) REF PLOTTER
```

Each picture drawn with the PLOTTER variable "stand lp plotter" is immediately printed via the standard output channel "standout", taking a page of its own, usually covering 119 character positions on 63 lines. Note that the store requirements must allow for the temporary creation of a [,]CHAR to map the printed page before printing it (say 2000 words).

Any other output channel opened with a CHARPUT variable can also be used for drawing pictures. A PLOTTER variable ("pv" say) must be declared, and the procedure "set lp plotter" is called to attach it to the CHARPUT variable. The procedure "set lp plotter" delivers the same PLOTTER variable "pv" so that it can be called at the parameter position of "draw picture" if desired. The full line width allowed by the CHARPUT will be used, less 1 if this is even, and the number of lines in a picture will be as given by "page size (cv)", less 1 if this is even, or 63 if the page size has not been set. An odd number of character positions and lines is used so that the centre of the picture area is always in the middle of a print position.

The scaling unit assumed for "natural scale" is one character separation width (1/10"), which is also 3/5 line separation, so the edges of the picture area are normally at (± 59.5 , ± 52.5) with the margins at (± 49 , ± 45). Normal lines are drawn by dots in the set of nearest character positions, with other distinctive characters for axis lines and dotted lines. The point plotting symbols are indicated by the characters . + X 0 = V @ # ↑ \$ & H M A W *

12.2.1 Line printer pictures offlined

A program producing line printer pictures can be reduced in size by offlining them. This is achieved by using the PLOTTER variable

```
PLOTTER lp off plotter;
```

which is declared in the segment "lp off plotter". The segment "lp plotter" must not be included if the program size is to be reduced. The pictures can be obtained afterwards in the manner described in section 12.1, by obeying the George macro-command

```
LPOFFPLOT
```

This counts as one program run of 1 minute mill-time and 30K core store.

* 12.3 The pen plotter

The pen plotter is a device which uses a ball point pen on a continuous roll of paper 75 cm wide. The pen can move across the paper at right angles to the direction of travel of the paper, and lines are drawn in any direction by suitable combinations of the two movements. The paper used may be either plain or pre-printed with centimetre/millimetre grid lines. As the pen is of the ball point type, it is not very good at drawing very short lines or point plotting symbols, and it is rather slow at moving the pen and paper and lifting the pen and lowering it, so the use of point plotting symbols and text characters, and dotted or broken lines should all be limited, and the drawing of individual lines and curves should be arranged to minimize the movement required between them as far as possible. Unless accurate scaling is required, it is usually more convenient to use the microfilm plotter.

The following are all declared in the segment "pen plotter":

```
PLOTTER pen plotter;  
PROC pen plot size = (REAL width, height)  
PROC pen plot outline = (BOOL yes or no)  
PROC pen plot paper = (INT paper type)  
INT plain paper  
INT mm squared paper
```

Pictures drawn with the PLOTTER variable "pen plotter" are offlined, to be drawn later by the pen plotter as directed by the computer operators. The picture area size is 34 cm x 24 cm by default, allowing two pictures to be drawn alongside one another across the paper. The scaling unit assumed for "natural scale" is one centimetre, and the standard character size gives a character spacing of 3 mm, so the side margins are 3.15 cm wide, and the top and bottom ones 2.7 cm. Each picture has a rectangle drawn round it and its number printed below to identify it, and a heading is drawn before the first picture to give the source of the pictures.

Pictures of other sizes can be drawn by calling "pen plot size" before the first picture of the new size, giving the required width and height in centimetres, these referring to the x and y coordinates respectively. Pictures too wide to be drawn across the paper are drawn along it, so the height of a picture may not be greater than the paper width. The rectangle drawn round pictures can be suppressed or reinstated by calling the procedure "pen plot outline" with parameter FALSE or TRUE respectively, before drawing the relevant picture, and the picture numbers can be altered or suppressed as described in section 8.

Plain paper is normally used, but cm/mm pre-printed paper can be used by calling "pen plot paper (mm squared paper)" at the start of the program. The procedure "pen plot paper" may also be called subsequently to change the type of paper to be used, but it will also reset all defaults, and should definitely not be called before each picture if the same type of paper is to be used. If pre-printed paper is used, the type of fitting used to fit a sketch to the picture area should normally be "natural scale" so that cm units are used. The centre of each picture will always lie at a cm grid point.

Graphical Output-Index

	<u>Section</u>	<u>Segment name</u>
+	2, 8.2	pictures
*	2, 6	pictures
/	2, 7, 8.1, 10	pictures, possibly plot number
↑	2, 6	pictures
'<>'	2	pictures
'->'	4, 4.2	sketches
'->:='	4, 4.2	sketches
add to graph file	11.1	graph file
arc	3	sketches
asterisk mark	3.1	pictures
axes	10	plot number
axis line	7.2	pictures
blank picture	2	pictures
blank sketch	2	
broken line	7.1	
broken line size	7.1	
case text	9.3	pictures
chain line	7.1	
chain line size	7.1	
circle	3	
circle mark	3.1	
closed curve	5.2	
cross in circle	3.1	
cross in diamond	3.1	
cross in square	3.1	
cross mark	3.1	
curve draw to	4.1	sketches
curve move to	4.1	
curved	5.1, 5.2	
diamond mark	3.1	pictures
dot in circle	3.1	
dot in diamond	3.1	
dot in square	3.1	
dot mark	3.1	
dotted line	7.1	
draw picture	8	
draw sketch	8.2	
draw to	4	sketches
end graphs	11.3	
equal scaling	8.1	
expand	6	sketches

Specific plotter segments are listed separately at the end of the index

LIBRARY

Graphical Output

Index

fit -- plu

	<u>Section</u>	<u>Segment name</u>
fit	8	
FITMOD	8.1	
fit range	8.1	pictures
fit sketch range	8.1	
FITTING	2	
frame number	8	
function draw to	4.1	sketches
function join	5.1, 5.2	sketch graphs
function move to	4.1	sketches
joined polygon	5.2	sketch graphs
label above	10	
label below	10	
label left	10	plot number
label right	10	
line	3	sketches
map line	6.1	
map triangle	6.1	
map vector	6.1	mappings
margin axes	8.2	
max curve error	4.1.1	pictures
max function error	4.1.1	sketch graphs
move to	4	sketch graphs
natural scale	8.1	sketches
number text	9.2	
offline plot	12.1	offline plotters
PICTURE	2, 8	
picture heading	8.2	
picture space	11.4	pictures
picture text	9.4	
plot	3.1	
plot number	9.2	sketches
plot string	3	plot number
plot text	9	sketches
PLOTTER	2	sketches
PLUS	2	
plus in circle	3.1	pictures
plus in diamond	3.1	
plus in square	3.1	
plus mark	3.1	

Specific plotter segments are listed separately at the end of the index

	<u>Section</u>	<u>Segment name</u>
RANGE	2	pictures
rectangle	3	sketches
REL	4	sketches
repeated sketch	6	pictures
retrieve sketch	11.5	pictures
rotate	6	sketches
set picture space	11.4	pictures
shear	6	sketches
shift	6	sketches
shift text	9	
SKETCH	2	pictures
sketch address	11.5	
sketch function	5.3	
sketch graph	5.1	sketch graphs
sketch points	5.2	
sketch trail	4	sketches
SKETCHMOD	2, 7	pictures
square mark	3.1	pictures
start graph file	11.1	graph file
start local graphs	11.2	local graphs
start trail	4	sketches
straight joins	5.1, 5.2	sketch graphs
stretch	6	sketches
stretch text	9.1	
string text	9	pictures
symbol	3.1	
tangent through	4.1	
TRAIL	4	sketches
trail arc	4.2	
TRAILMOVE	4	
TRANSFORMATION	2, 6	pictures
unbroken line	7.1	pictures
unjoined	5.1, 5.2	sketch graphs
whole area	8.1	pictures
x axis	10	plot number
x list axis	10	plot number
x log scale	6.2	mappings
xy log scale	6.2	mappings
y axis	10	plot number
y list axis	10	plot number
y log scale	6.2	mappings

Specific plotter segments are listed separately at the end of the index

LIBRARY
Graphical Output
Index of Plotters

Segments for specific plotters

<u>Segment name</u>		<u>Contains</u>	<u>Associated George command</u>
lp plotter	12.2	set lp plotter stand lp plotter	
lp off plotter	12.2.1	lp off plotter	LPOFFPLOT
pen plotter	12.3	pen plotter pen plot outline pen plot paper pen plot size plain paper mm squared paper	
* micro plotter	12.4	micro plotter cine plotter mp thin line	
* coragraph	12.5	coragraph cora plot size cora cut cora draw cora scribe	CORAPLOT

*This plotter is only available at RRE.

1 SIMPLE INPUT/OUTPUT

In the simplest form of single-channel input, data is placed after the EXECUTE command in the Job Description or filed separately (GEORGE 2.1, 2.3) and is input to the program by calls of the procedure "read". The data may be from cards or from paper tape using the standard tape code (as produced by Olivetti teleprinters), with the proviso that for simple input there should be less than 160 characters on each line of data (see section 2.5 if there are more). For single-channel output, results are printed by calls of the procedure "print", and appear automatically on a line-printer document.

An introduction to Input/Output (or 'transput' for short) is given in "Algol 68-R Users Guide", 2nd edition HMSO (July 1974), which contains much of the material from sections 1 and 4 of this part of the manual.

1.1 The procedures read and print

The procedure "read" has one parameter only, but this can take a wide variety of forms. It can be a variable, or a layout procedure or a bracketed list of such items, which need not be all of the same kind. As examples,

read(r)	reads an appropriate object into r
read(newline)	moves the reading position to the start of the next line or card
read((x,i,newline))	is equivalent to read(x); read(i); read(newline)

The objects in the data stream must be appropriate to the modes of the variables. If r refers to an array of 6 integers, the "appropriate object" is a succession of 6 integers. If it is a structure with an integer field and a character field, the data must be an integer followed by a character. The identifier "newline" is the name of a library layout procedure.

The procedure "print" works in a similar way to read, except that items are layout procedures or expressions which deliver printable objects. If an expression is simply a variable, de-referencing takes place automatically, but no de-referencing takes place within arrays or structures. As an example,

```
INT i := 4;  
STRUCT(INT head, REF INT tail) s := (6,i);  
print((head OF s, tail OF s))
```

prints 6 and, after de-referencing, 4. However, print(s) would be illegal because the tail OF s cannot be de-referenced. References cannot be printed.

The styles for reading and printing described in sections 1.2 and 1.3 are mutually compatible.

1.2 Standard forms for input items

A single item of input data may be a REAL, COMPLEX, INT, LONG INT, BITS, BOOL, CHAR, BYTES, LONG BYTES or LONG LONG BYTES. Arrays of such items, and character strings, are treated in section 1.2.3. Most values in a data stream are composed of several characters, so it is important to know the rules which decide where one item ends and the next begins. These rules are particularly simple for numerical values, several of which can be written on the same line or card provided that they are separated by one or more spaces. The end of the line must not split a value, as it is an alternative way of indicating the end of one value and the start of the next. The general principle is that the read procedure will read as many characters as it needs to complete a value. It does this by looking to see if the next character would be a legal continuation or not. If not, the value is terminated and the character which would have been illegal is left unread. However, it is not discarded; it becomes the first character of the next item. For example, if the data stream contained the integers 24 and 25 separated by one space, the space would indicate the end of 24 and be read as the first character of the value 25. Preliminary spaces and new lines are ignored for numerical values, so any amount of layout can be placed between such items.

Especial care is necessary when reading literal character sequences, that is to say items of modes BYTES and []BYTES, including the LONG forms, STRING and []CHAR. For these items, spaces are read as characters and all the characters of the item must be on the *current line or card*. This rule has an important consequence, for the reading position will very often be at the end of a line after reading an item of data. It will never be at the beginning of a new line after reading in a value. If our literal character sequence is on the next line, the reading position does not go to the new line automatically as it would for a numerical value. The position must be moved to the start of the new line explicitly, by including 'newline' at the appropriate place in the variable-list or by writing read(newline) as a separate call.

1.2.1 Input of numbers

In the following, curly brackets indicate components which may optionally be omitted. "Layout" means spaces, new lines or new pages.

REAL

All layout is ignored until the first non-space character is read. This must be a sign or a digit.

Form {sign}decimalnumber{&}{sign}digits}

where decimalnumber is a sequence of digits possibly including a point. The symbol "&" means "times 10 to the power". Spaces may optionally be included after +, -, or &. Elsewhere a space or new line will terminate the number. The limits of a real are approximately 5&-78 to 5&76.

Examples 123 123&4 123.4 -123.45&-6

COMPLEX

Form REAL{layout}?{layout}REAL

Example 123.4&-5 ? -23.4&-5

INT

All layout is ignored until the first non-space character is read. This must be a sign or a digit.

Form {sign}{spaces}}digits

Examples 8388607 (largest positive INT)
-8388608 (most negative INT)

LONG INT

As for INT, with limits $\pm(2^{46}-1)$.

BITS

All layout is ignored until the first non-space character is read. This must be a digit 0 or a digit 1.

Form Up to 24 0's or 1's

If less than 24 digits are given the BITS value is made up with leading zeros.

Example 10000000 (the BITS representation of the integer 128)

1.2.2 Input of non-numerical items

BOOL

All layout is ignored until the first non-space character is read. This must be T (for TRUE) or F (for FALSE).

CHAR

If the reading position is at the end of a line or page, read takes a new line or page first, though not for arrays of characters (see 1.2.3). The character is then read. Space is a character but newline is not.

Form One character from the following set

0	1	2	3	4	5	6	7	8	9
:	;	<	=	>	?	sp	!	"	#
£	%	&	'	()	*	+	,	-
.	/	@	A	B	C	D	E	F	G
H	I	J	K	L	M	N	O	P	Q
R	S	T	U	V	W	X	Y	Z	[
\$]	↑	←						

BYTES

If the reading position is at the end of a line (or page), read does not take a new line (or page) first. This must be done independently, for example by read(newline).

Form 4 characters

Example AN 8

LONG BYTES

As for BYTES, but 8 characters

LONG LONG BYTES

As for BYTES, but 12 characters.

1.2.3 Input of Arrays

Note on terminology: The terms fixed bounds and non-flexible bounds are synonymous, and apply to arrays whose size cannot be changed within the serial clause in which they are declared, except when reading rows of characters, as described below. Arrays with flexible bounds contain in their declaration the word FLEX, and use the heap. The terms row and one-dimensional array are synonymous.

When an array variable is used in the parameter of read, the procedure normally reads an appropriate sequence of items on the data stream so as to fill the array. For example,

```
[1:2,1:2,1:2]INT q;  
read(q)
```

reads 8 integers, which are assigned to q with indices increasing in lexicographic order (111,112,121,122,211,212,221,222). If the array variable has flexible bounds, then except for arrays of characters, the actual bounds currently in force will determine the number of items read.

Arrays of any of the items listed in sections 1.2.1 and 1.2.2 can be read, but special rules apply to reading rows of characters, as described below.

ARRAY OF CHARACTERS (FIXED BOUNDS)

New line or page must not occur before or within the sequence of characters to be read. If the previous item of data was terminated by a new line, this must first be read independently, for example by read(newline). The number of characters read is normally determined by the bounds of the array, but for simple row variables, it is possible to read a shorter sequence by ending it with a new line. This acts as a terminator, though the reading position does not move past it. The upper bound of the row-of-characters variable will then be automatically changed to reduce the size of the array to suit the number of characters read. The original size of the array can be restored only by re-declaration. Note that if a new line terminator is encountered before any characters are read, the size of the array will be reduced to zero.

STRING

This is a standard mode synonymous with a row of characters with flexible bounds, and makes use of the heap. When reading into a string variable, the sequence of characters is terminated by a new line, which is not itself read. The size of the array is then reduced or enlarged to suite the number of characters read. In some practical situations, the inconvenience of a fixed-bounds variable may be preferred to the cost of using the heap with STRING.

LIBRARY
Input/Output
1.3

1.3 Standard form of printing

1.3.1 Items printed with automatic layout safeguards

For the following items, if there is not enough room left on the line, a new line is taken automatically before the item is printed.

REAL

Except at the beginning of a line, each complete number is preceded by one space. The length is 18 characters, or 17 at the start of a line.

Examples +1.0123456789& +0
 -1.0123456789&-12

COMPLEX

Except at the beginning of a line, the complex number is preceded by one space. The length is 37 characters, or 36 at the start of a line.

Example +1.0123456789&+12 ?+1.0123456789& -2

INT

Except at the start of a line, the integer is preceded by one space. The length is 9 characters, or 8 at the start of a line, of which up to 7 are digits.

Examples -8388608
 +8388607
 -123
 +0

LONG INT

As for INT, but 7 digits longer.

BITS

Except at the beginning of a line, the item is preceded by one space. Then 24 binary digits (0 or 1) are printed consecutively.

BOOL

Except at the beginning of a line, the print of T (for TRUE) or F (for FALSE) is preceded by one space.

CHAR

After the automatic new line if necessary, the character is printed. The length is 1 character.

1.3.2 Items with no layout safeguards

No new lines are taken automatically when the following items are printed, and overshooting will cause a fault.

BYTES

Form 4 characters and no layout

Example 100%

LONG BYTES

As for BYTES, but 8 characters

LONG LONG BYTES

As for BYTES, but 12 characters

1.3.3 Printing of arrays

The elements of the array are printed in lexicographic order of indices, ie for an array with bounds [1:2,1:4] the order of printing would be the elements with indices (11,12,13,14,21,22,23,24).

ARRAY OF REAL,COMPLEX,INT,LONG INT,BITS,BOOL

Each element is printed in the appropriate form given in 1.3.1. A new line is automatically taken if there is not enough room on a line to print the next element.

ARRAY OF CHAR

The sequence of characters is printed with no added layout. Overshooting a line will cause a fault.

STRING

As for any array of characters (above).

ARRAY OF BYTES, LONG BYTES, LONG LONG BYTES

No additional layout is inserted between elements and overshooting a line will cause a fault.

1.4 Spaces, new lines and new pages

This section describes the library procedures space, backspace, newline and newpage when used as parameters of print and read.

1.4.1 Space and backspace

A call of "space" as a (procedure) parameter, e.g.

```
print(space)
```

moves the printing position ahead by one character without printing. As each new line of output is initially filled with spaces, then (providing that no backspacing has been performed) the effect of print(space) is the same as that of print(" "), which physically puts a space character at the current printing position in the output file.

As a parameter of read, the effect of space is to move the position forward by one character without reading. As an illustration, let i be an integer variable and ch be a character variable, and suppose the data about to be read is

365AB

The effect of read((i, ch)) would be to assign 365 to i and "A" to ch, but the effect of read((i, space, ch)) would be to assign 365 to i and "B" to ch.

The reading of backspace moves the reading position backwards by one character. With the data 365AB, the effect of read((i, backspace, ch)) would be to assign 365 to i and the character "5" to ch. Difficulties may occur at the end of a line; see the last paragraph of section 7.2.

As a parameter of print, the use of backspace moves the printing position back one character and causes the next printed character to replace what was previously there. Thus print (("ABC", backspace, "DE")) prints ABDE.

1.4.2 Newline and newpage

A call of print(newline) takes a new line of output and print(newpage) a new page. New pages are not provided with headings. The maximum length of a line is 120 characters, and the maximum practical number of lines between perforations on line-printer output is 60. For input, the effect of read(newline) is to move the reading position to the start of the next line or card, as may be necessary before reading a row of characters on the next line or card, or for skipping over comments which have been written at the ends of lines of data. A call of read(newpage) may be useful when reading (paged) data which has previously been generated by calls of print. It moves the reading position ahead to the start of the next page.

1.5 Changing printing styles

The standard form of printing usually produces results which are difficult to read, as new lines are only taken where necessary and each number is printed to the maximum number of digits. To print numbers in special styles with varying numbers of figures, tabulating them and possibly including headings, the system of *formatted transput* should be used. This is described in the "Algol 68-R Users Guide" (2nd edition, HMSO July 1974), and also in section 4 of this part of the manual.

1.5.1 Numberstyle

For a simple alteration to just one aspect of the standard layout, the library structure "numberstyle" may be useful. Most of the integer fields of numberstyle are used to control the printing of numbers and are preset to values which correspond to the style of printing described in section 1.3.1. If different integers are assigned in the program, the style can be changed. For example, the assignment

```
int OF numberstyle := 3
```

restricts integer print-outs to 3 digits. Whenever a new style is set up, it applies to all printing from then onward unless the procedure "standput" is used (see section 3.4). The fields of numberstyle are as follows:

<u>field</u>	<u>standard setting</u>	<u>meaning</u>
spaces	1	number of spaces output before a number which is not at the start of a line
sign	27	code for character used for "+" when positive numbers are output 27 gives + 29 gives space 0 gives void, and makes output of negative numbers illegal
		Other settings for this field are described in section 1.5.3
int	0	standard setting which gives 7-digit integers, 14-digit long integers and 24-bit bits values. If this field is set to n, the number of digits for an integer or long integer will be limited to n, and bits values will be printed as n-digit integers.
before	1	number of digits before the point when a real is printed
after	10	number of digits after the point when a real is printed: if set to 0, the point is also suppressed
exp	2	number of digits output in exponent of a real number: if set to 0, the "&" is also suppressed

1.5.2 Reading numbers containing gaps

The standard form for number input described in 1.2.1 does not permit the use of spaces for grouping of digits, eg

-1 234 567

This example would be read as 3 separate integers. To prevent such spaces from acting as terminators, the number of consecutive spaces required to terminate a number can be increased by assignment to the field "gap OF numberstyle". Thus,

gap OF numberstyle := 2

makes the second of 2 consecutive spaces a terminator, so that a single space within or after a number would be passed over.

1.5.3 Further settings of "sign OF numberstyle"

The field "sign OF numberstyle" controls three aspects of printing numbers and one aspect of reading. For printing it determines the sign indication used as described in 1.5.1, the treatment of leading zeros of numbers (normally these are replaced by leading spaces) and the base to which integers and long integers are printed (normally decimal). For input it determines the base (normally decimal) for reading integers and long integers. These normal conventions can be changed by assigning values made up from those given in the next sections to "sign OF numberstyle". Combinations of alternatives are obtained by adding together the settings given. For example, to output numbers without leading spaces or leading zeros, retaining the character "+" for positive values, the setting required is 283 (256 + 27). To print unsigned integers in octal, including leading zeros, the setting would be 8r00030300.

Standard conventions for the base of input and output and the treatment of leading zeros can be restored by setting sign OF numberstyle back to one of the values given in 1.5.1.

For alterations of any complexity, it is preferable to use the 'frames' of formatted transput which provide the same facilities, rather than change numberstyle.

1.5.3.1 Printing of leading zeros and spaces

With the normal conventions for printing, a number is "right justified" within the number of digits being printed, that is, spaces are printed instead of leading zeros of the number. Alternative styles of printing can be obtained by using the d, sv and wv 'frames' of formatted transput as described in Input/Output 4.6, or by setting "sign OF numberstyle" as follows.

<u>Setting of sign OF numberstyle</u>	<u>Effect</u>
192 ie 8r0000 0300	All leading zeros are printed
256 ie 8r0000 0400	Leading zeros are omitted and are not replaced by spaces
640 ie 8r0000 1200	Leading zeros are replaced by trailing spaces ie the number is "left-justified"

In all these cases the integer 0 will be printed with 0 in the units position.

Examples

In the following examples the normal sign indication (27) has been added to each of the above settings, and the standard setting of other fields of numberstyle are assumed.

<u>Value of sign OF numberstyle</u>	<u>Output from print ("[", 123, "]")</u>
27	[+123]
219	[+0000123]
283	[+123]
667	[+123]

1.5.3.2 Printing to non-decimal bases

This can be obtained by using the formatting symbols described in Input/Output 4.7 or by the following settings of sign OF numberstyle. For representations to base 16, the extra characters used in printing are the letters A to F inclusive. Negative integers do not cause a fault but are printed without a sign as though 2⁴ had been added (ie the binary sign digit is incorporated into the most significant digit). Positive integers are printed with spaces instead of leading zeros unless a setting from section 1.5.3.1 is added to the values below. When using a non-decimal base in this way the field "int OF numberstyle" must also be set to the corresponding value shown in the table.

<u>Base for printing</u>	<u>sign OF numberstyle</u>	<u>int OF numberstyle</u>	<u>how negative integers -2²³ to -1 are printed</u>
2	8r0001 0000	24	100000000000000000000000 to 111111111111111111111111
4	8r0002 0000	12	20000000000 to 33333333333
8	8r0003 0000	8	4000000 to 7777777
16	8r0004 0000	6	800000 to FFFFFF

When printing long integers, int OF numberstyle should be set to twice the above values. Negative long integers are printed without a sign as though 2⁴ had been added (long integers are held in the computer to 47 binary digits including the sign).

If a sign indication (say 27) is added to the above settings of sign OF numberstyle, integers and long integers will be printed to the non-decimal base indicated, but with conventions similar to those used in the scale of 10. The numbers will be treated as signed and will be printed to n digits, where n is determined by int OF numberstyle.

1.5.3.3 Reading integers using non-decimal bases

To read integers (but not long integers) using a non-decimal base, with the binary sign digit incorporated, the settings for sign OF numberstyle are:

<u>Base for input</u>	<u>sign OF numberstyle</u>
2	8r0100 0000
4	8r0200 0000
8	8r0300 0000
16	8r0400 0000 extra characters are letters A to F inclusive

However, if a sign is present in the data, both integers and long integers are read as signed numbers with conventions similar to those used in scale of 10.

1.5.4 NUMBERSTYLE variables

The library variable "numberstyle" has mode REF NUMBERSTYLE, and for repeated changes to numberstyle, a convenient plan is to declare NUMBERSTYLE variables in the program. The following example should be self-explanatory:

```
NUMBERSTYLE style0 := numberstyle,  
            style1 := numberstyle,  
            style2 := numberstyle;  
  
            int OF style1 := 3;  
            int OF style2 := 5;  
  
----  
----  
numberstyle := style1; print(123);  
----  
numberstyle := style2; print(12345);  
----  
numberstyle := style0; print(1234567);  
----  
----
```

In this particular example, there is no saving in convenience, but if several fields of numberstyle are to be changed simultaneously, the method may be found worthwhile. Another method, using procedure parameters of "print", is described in section 3.4.

A common error is to attempt a change of numberstyle in the actual parameter of print by using a serial clause as follows:

WRONG print((1234567, int OF numberstyle := 3; 123))

This is wrong because all the clauses delivering objects to be printed are obeyed before the procedure itself is obeyed. The above call would obey the assignment to numberstyle before attempting to print 1234567, and would therefore fault.

1.6 Fault displays

END OF FILE	applies to input files and signifies an attempt to read beyond the end of the file
END OF LINE	normally occurs when a row of characters is being printed, and will not fit into the line. When rows of characters are printed, no new lines are taken automatically. The fault can also occur during input if a new line is encountered in an illegal position, e.g. after "&" in a REAL number
NUMBER FORMAT ERROR	occurs when an input integer or real overflows. It also occurs if an output number is too big or wrong for the numberstyle set. The number is then printed in the default numberstyle before the program faults, an overflowed number being preceded by the character ">".
ILLEGAL CHARACTER	occurs if an unexpected character is met during input, e.g. a letter when a number is about to be read

The above are the most commonly occurring faults at run-time; the message is displayed on the monitoring file.

2 CHARPUT VARIABLES

For single-channel input and output, the simplest scheme is to use "read" and "print" as described in section 1. These procedures hide the existence of charput variables, which need not be declared in the user's program unless extra channels of input or output are required. However, to gain facilities beyond those described in section 1, it is necessary to use charput variables whether extra channels are needed or not.

Each channel of input or output uses its own charput variable mainly to keep the place in the data file, much as a tape reader keeps the place on a tape. In multi-channel work (sections 2.1 to 2.5), the desired channel is always selected in the program by specifying its particular charput variable. The procedures read and print use the variables "standin" and "standout" which are library charput variables for the standard input and output channels. Extra charput variables must be declared by the user himself, as described in sections 2.2 and 2.3.

2.1 Procedures "get" and "put"

The general input and output procedures are get and put, which differ from read and print only in that they take an extra parameter which is the charput variable for the required channel. `read(x)` is exactly equivalent to

```
get(charput variable, x)
```

and `print(x)` is exactly equivalent to

```
put(charput variable, x)
```

when the charput variables are written as "standin" and "standout" respectively. As an example,

```
read(x);  
print(("SIN(X) =", sin(x), newline))
```

could alternatively be written as

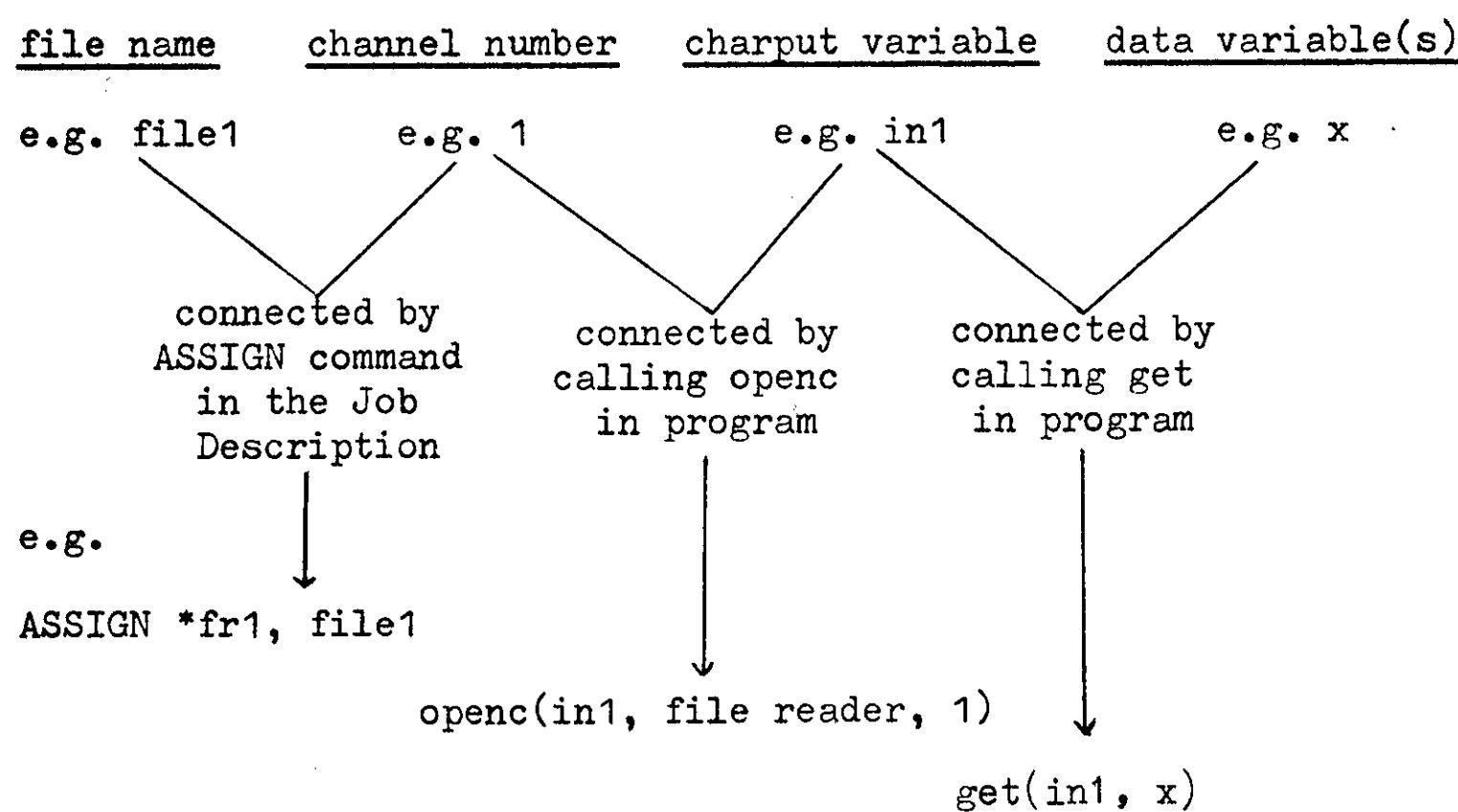
```
get(standin, x);  
put(standout, ("SIN(X) =", sin(x), newline))
```

2.2 Extra channels of input

A program may need to access more than one set of data. Each set must be input to a George file, as summarized in GEORGE 2.2, and explained in greater detail in GEORGE 5. As the names of George files are not used in programs, a connection must be established between the file names and charput variables which will be used for channel selection in the program. The charput variables can be named arbitrarily; if we choose in1, in2 and in3 as charput variables for 3 extra input channels, they are declared by writing

```
CHARPUT in1, in2, in3;
```

The connection between the file names and these charput variables is effected partly in the Job Description and partly in the program. The Job Description must contain ASSIGN commands which associate file-reading channels, numbered from 1 upwards, with the data files (see GEORGE 4.2.4). The program must call the channel-opening procedure "openc" as described in section 2.4 to associate the channel number with a charput variable. Diagrammatically,

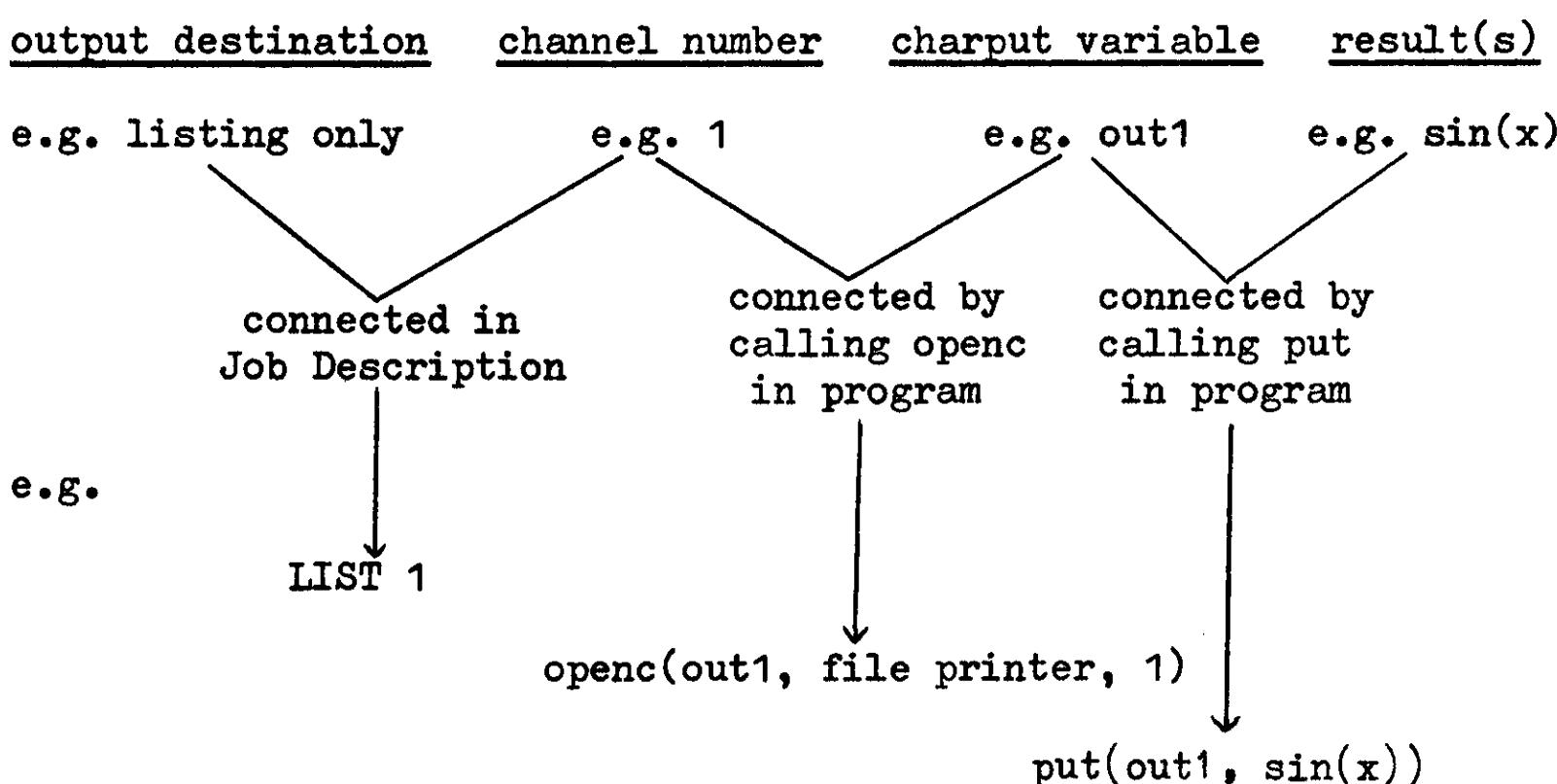


2.3 Extra channels of output

The destinations of the results on different channels may either be temporary files if printed output documents are all that are needed, or permanent files if the results are to be kept indefinitely. In either case, the extra output channels must be numbered from 1 upwards and connected to the files by George commands such as LIST or ASSIGN in the Job Description (see GEORGE 4.2.5). In the program, a charput variable must be declared for each extra channel of output, e.g.

```
CHARPUT out1, out2, out3;
```

Channel numbers are associated with these charput variables by the channel-opening procedure "openc" (section 2.4).
Diagrammatically,



2.4 Opening and closing channels

The purpose of "opening" a channel is to initialize a charput variable for input or output and to associate it with an actual channel number (see diagrams of 2.2 and 2.3). This is effected by a call of the library procedure "openc". For example, to open two extra channels of input, 1 and 2, and two extra channels of printed output, 1 and 2, the program must contain

```
CHARPUT in1, in2, out1, out2;
openc(in1, file reader, 1);
openc(in2, file reader, 2);
openc(out1, file printer, 1);
openc(out2, file printer, 2);
```

before the channels are used with get and put. The first parameter of openc is the charput variable, the second is a library identifier giving the type of channel required and the third is the input or output channel number. There are two simple channel types: "file reader" for input and "file printer" for output. A file reader channel can be used for reading any sort of file (graphic or normal) provided that there are less than 160 characters (including any shift characters) on each line of data. A file printer channel can be used for printed output with up to 120 characters per line. Special types of channel (described in 2.5) are needed for longer lines on input or output, for any output destined for paper tape, and for output of characters other than those in the graphic set.

It is advisable to open channels immediately after declaring their charput variables, as it is necessary that the declarations and calls of openc be in the same nesting level of the program. Once a channel has been opened, it stays open until the program ends or until it is closed by a call of "closec". This takes only one parameter, the charput variable, eg closec(out2), and has the effect of outputting any half-finished line of output and then releasing the file attached to the channel. Since all files are automatically released at the end of a program run there is usually no need to close channels. Instead, for an output channel, printing should end with a call of newline, otherwise the output buffer will not be emptied and the very last line of results will be lost.

The standard channels used by the procedures "read" and "print" each have channel number 0. These channels are opened automatically in a special way to enable them to be connected to a MOP terminal, so channel number 0 should never be explicitly mentioned in a program. The charput variables "standin" and "standout" used by read and print can however be used in a call of openc (with another channel number) if a different type of channel is needed. The output buffer of standout is always emptied automatically at the successful completion of a program run.

2.4.1 An example of multi-channel input/output

Below is an outline sketch of a program which uses two channels of input, the standard channel and one extra, and two of output similarly.

example

```
BEGIN
    CHARPUT in1, out1;
    openc(in1, file reader, 1);
    openc(out1, file printer, 1);
    REAL x0, x1, y0, y1;
    read(x0);
    get(in1, x1);
    y0 := etc ;
    y1 := etc ;
    print(y0);
    etc ;
    put(out1, (y1, newline))
END
FINISH
```

The Job Description must contain the links between channel numbers and actual sources of input data or destinations of output results. The minimum arrangements for the above example would be:

Standard input channel	By default, the calls of "read" will use data embedded in the Job Description after the EXECUTE command
Channel 1 of input	There must be an ASSIGN command in the Job Description to connect this channel with a data file
Standard output channel	By default, results from calls of "print" will be put out on a line-printer document
Channel 1 of output	The command LIST 1 in the Job Description would cause these results to be printed on another (separate) line-printer document (and not kept in a permanent file).

2.5 Use of special channel types

The simple channel types "file reader" and "file printer" are described in section 2.4. Different types of channel are needed for the special purposes described in the following sections. Remember that for each call of openc a George ASSIGN or LIST command is needed in the job description (GEORGE 4.2.4, 4.2.5) - its exact form will depend on the type of channel.

2.5.1 Long lines of printed output

The standard file printer channel allows up to 120 characters per line. To increase this use a "wide printer" channel as in

```
openc (out1, wideprinter (160), 1)
```

The parameter of wideprinter is the maximum line length required, and must be less than 511. The line printers in use at RRE have 160 printing positions per line.

2.5.2 Data in long lines (few line feed characters)

Every file is made up as a series of lines, a complete line being formed during INPUT whenever a standard line feed (LF) character is encountered. A "file reader" channel can only deal with up to 160 characters per line (including any shift characters inserted when the INPUT is NORMAL). This can be increased to a maximum of 510 by using a wide reader channel, as in

```
openc (in1, wide reader (200), 1)
```

If line feed characters on the original tape are irregular or non-existent, the file must be read via a "tape reader" channel. This can be the standard input channel (automatically a "tape reader") or an extra channel opened with "tape reader" in place of file reader in the call of openc. Long lines in the file are then automatically split up into shorter ones before being passed to the program. The divisions are made at arbitrary points so that the end of a line may occur at an unexpected place in the data, possibly in the middle of a number. This would cause "read" or "get" to terminate the number prematurely: to avoid this ends of lines must be ignored in the program as shown in section 6.2.

2.5.3 Reading and writing non-graphic-set characters

The graphic set characters are those of mode CHAR. No special channel is needed to read other characters, but the basic input/output procedures described in section 6 must be used. When non-graphic set characters are to be output, shifts need to be inserted. This is done by the library procedures for an output channel opened with "file punch" in place of file printer. No more than 128 characters (including shifts) may be output between calls of newline in the program. Note that the standard library procedures do not produce any non-graphic set characters, in particular, carriage return; such characters must be output explicitly by the procedure "out OF" described in 6.3. When a file is output on paper tape by LISTFILE a line feed and 3 blanks will be punched on the tape wherever newline was called in the program. For this reason a file punch channel is usually unsuitable for output destined for paper tape.

2.5.4 Output on paper tape

All paper tape is produced by a LISTFILE command for the appropriate file. At the end of each line in the file, a line feed and 3 blanks are punched on the paper tape (or just line feed if the file is 'allchar'). For a file printer or file punch channel a line in the file is produced whenever newline is called in the program, so unwanted line feeds may be produced on the tape. To overcome this a channel opened as a "tape punch" should be used. For a tape punch channel George forms lines in the file only when the program produces an actual line feed character from "out OF"; several lines in the program (produced by "newline") may thus make up one line in the file. A line in the program must not exceed 128 characters (including shifts), so new line must be called regularly - the easiest way to do this is shown in section 6.3. The final paper tape output will contain only those characters explicitly produced by the program.

2.5.5 Input from and output to []CHAR variables

The full facilities of input and output can be directed to read from and print to []CHAR variables rather than files. The transput is controlled as usual by a charput variable, say v, initialised by

```
openc (v, string reader, 1)      for input  
or    openc (v, string printer, 1)   for output
```

The channel number is not relevant - since no files are involved there is no need for an associated George command. The actual array to be used is associated with the charput by calling

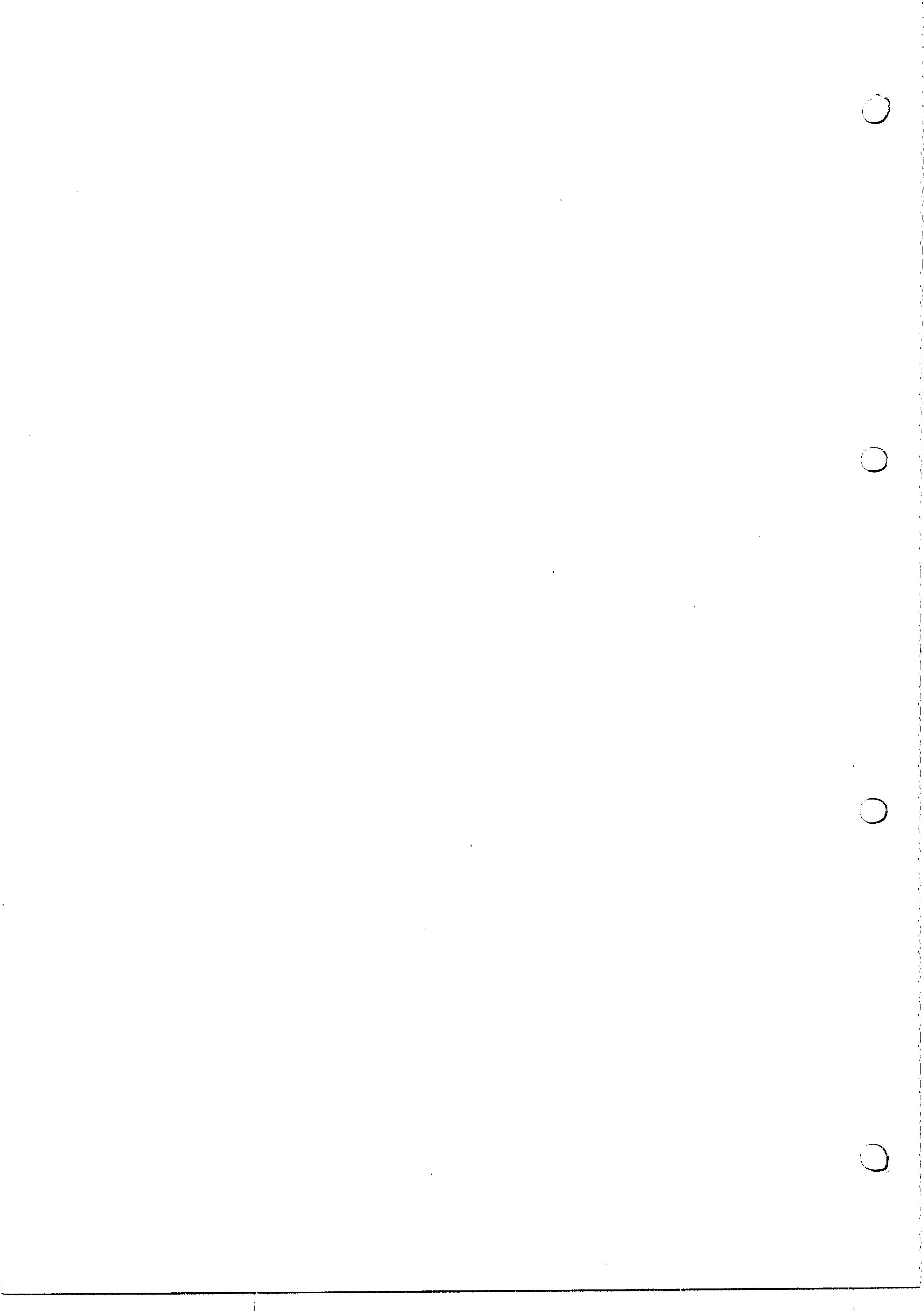
```
PROC next string = (REF CHARPUT v, REF [ ]CHAR aa)
```

Transput controlled by v will then treat aa as one line; for output aa will initially be cleared to spaces. Subsequent calls of new line or newpage will cause transput to start again from the beginning of aa; again, for output, the array will first be cleared to spaces. The length of aa must not be greater than 510 characters, and its bounds should not be altered (by using it as a parameter of read for example) after calling next string. A slice of an array may be used, provided that its elements are contiguous (eg it is a row rather than a column of a two-dimensional array).

It is possible to output to a channel opened as a "string reader"; this avoids the clearing of the []CHAR variable on each newline or nextstring. (It is also possible to read the 'current line' of output, ie characters written since the last newline or nextstring, as described in section 7.2.)

The procedure nextstring may be called many times, to associate different (or even the same) []CHAR variables with the charput. Typically, the new line event (see section 5.3.5) may be altered to associate a new row of characters automatically on each new line (by calling nextstring, and delivering zero as the result of the event procedure). Section 5.3.5.1 gives more details of this, and an advanced example is given in section 7.3.

For the full character set, "string lector" and "string punch" are provided. These deal with (or produce) shift characters; they otherwise behave exactly as string reader and string printer respectively.



3 LAYOUT FACILITIES USING CHARPUT VARIABLES

3.1 Library layout procedures

The procedures space, backspace, newline and newpage have been described in section 1.4 as parameters of read and print. These procedures may also be used by calling them directly, if they are provided with a charput variable as parameter. For instance,

```
space(standout)
```

is a call of the procedure "space" on the standard output channel, which has standout as its charput variable. The above call has exactly the same effect as

```
print(space)
```

or as

```
put(standout, space)
```

but space(standout) is more direct and therefore gives a more efficient program. For the procedure to apply to one of the user's extra channels, the parameter would be the charput variable associated with that channel. Charput variables other than standin and standout must be declared in the user's program as explained in section 2.2.

LIBRARY
Input/Output
3.1

PROC setcharnumber = (REF CHARPUT v, INT n)

A call of setcharnumber(v, n) moves the reading or printing position on the channel controlled by the charput variable v to character position n. The movement may be forwards or backwards, and the next character to be read or printed will be the nth on the line. This procedure cannot be used as a procedure parameter of read or print, as it has two parameters. See also section 7.2.

PROC space = (REF CHARPUT v)

A call of space(v) moves the reading or printing position on the channel controlled by the charput variable v forward by one character without actually reading or printing a character. The upper limit for the number of characters which can be printed on one line is normally 120. See also section 7.2.

PROC backspace = (REF CHARPUT v)

A call of backspace(v) moves the reading or printing position on the channel controlled by the charput variable v back one character. See also section 7.2.

PROC newline = (REF CHARPUT v)

A call of newline(v) moves the reading or printing position on the channel controlled by the charput variable v to the start of the next line or card. If an assignment has been made to pagesize(v), calls of newline on this channel will automatically call the procedure newpage at regular intervals (see section 3.2, PROC pagesize).

PROC newpage = (REF CHARPUT v)

If v is the charput variable for an output channel, newpage(v) takes a new page of output. It does not supply a page heading. If v controls an input channel, and if the input data contains new pages (having previously been generated by output), newpage(v) moves the reading position to the start of the next page.

3.2 Positional information

PROC page number = (REF CHARPUT v)INT

For the input or output channel controlled by the charput variable v, page number(v) delivers the number of the current page. The first page is numbered 1.

PROC line number = (REF CHARPUT v)INT

For the input or output channel controlled by the charput variable v, line number(v) delivers the number of the current line. The number of the first line on each page is 1.

PROC charnumber = (REF CHARPUT v)INT

For the input or output channel controlled by the charput variable v, charnumber(v) delivers the character position of the next character to be read or printed. After a call of newline(v), this is position 1.

PROC charsleft = (REF CHARPUT v)INT

For a "file printer" or standout channel, charsleft(v) delivers the number of character positions which remain available for printing. The procedure is particularly useful for testing whether there is room on a line for printing a row of characters.

PROC pagesize = (REF CHARPUT v)REF INT

For an output channel controlled by the charput variable v, pagesize(v) delivers a reference to the number of lines per page. An assignment such as

pagesize(w) := 50

sets the number of lines per page on the channel with charput w to 50. Thereafter, whenever 50 lines have been printed on a page on this channel, the procedure newline will call newpage automatically. If no assignment to pagesize is made, there are no automatic calls of newpage. For line-printer output, the number of lines per page should not exceed 60. The procedure can also be used to determine the current setting of the page size - if this should for some reason be unknown. As an example

```
IF line number(w) > pagesize(w) - 5
THEN newpage(w)
FI
```

would safeguard against an unwanted new page coming in the middle of an impending 5-line object of data.

3.3 Constructing layout procedures for use in read and print

The procedure "read" (and hence also get) may have as a parameter either a variable into which an object is to be read or it may have the name of a layout procedure. Similarly, the procedure "print" (and hence also put) may have as a parameter either an object to be printed or the name of a layout procedure. Such layout procedures must be of mode PROC(REF CHARPUT), or in other words must take a single parameter which is a charput variable.

When the name of a layout procedure is given as a parameter of (say) put, the effect is not to 'output' the procedure - which would be meaningless - but to obey it with the charput variable used in the call of put as its actual parameter. For example, the effect of

```
put(standout, (x, space, y))
```

is to output the object x, obey space(standout), and output the object y, in that order. In calls of read or print, the charput variable supplied to the layout procedure is standin or standout respectively.

Non-standard layout procedures can be declared in a program; provided that they take a charput variable as their one parameter, they can be used as procedure parameters of read, print, get and put in the same way as library procedures like space. The following trivial illustration shows the pattern:

```
test layout
BEGIN
    PROC tenspaces = (REF CHARPUT v):
        (TO 10 DO space(v));
        print(("A", tenspaces, "B"))
    END
FINISH
```

The actual parameter standout for tenspaces is automatically supplied by the procedure "print".

3.4 The procedure "standput" and re-setting of numberstyle

When numberstyle has been changed as described in section 1.5, its new setting applies to all future printing of numbers through the procedures print or put. However, the standard style can be obtained at any time, without having to restore the standard values to the various fields of numberstyle, by use of the procedure "standput", which takes parameters like put. For example,

```
sign OF numberstyle := 29;  
standput(standout, 1000000);  
print(1000000)
```

outputs +1000000 1000000

A convenient system for changing numberstyle between various non-standard settings within the parameter of print itself is to declare 'layout procedures' as described in 3.3, even though they neglect their compulsory charput parameters. The example shown in 1.5.4 may be re-programmed as follows:

```
PROC style0 = (REF CHARPUT v):  
    (int OF numberstyle := 7);  
  
PROC style1 = (REF CHARPUT v):  
    (int OF numberstyle := 3);  
  
PROC style2 = (REF CHARPUT v):  
    (int OF numberstyle := 5);  
  
----  
----  
  
    print((style1, 123, style2, 12345, style0, 1234567));
```

None of the above procedures uses its REF CHARPUT parameter, which is only there to give the layout procedure the correct mode.

3.5 Reading rows of characters - arbitrary terminators

The simple rules for reading one-dimensional arrays of characters are given in section 1.2.3. Briefly they are:

If the array has non-flexible bounds, characters are read until the array has been filled or a terminator is encountered; in the latter case the size of the array will be automatically reduced to suit the number of characters read. If the array has flexible bounds, characters are read until a terminator is encountered; the size of the array is then reduced or enlarged to match the number of characters read.

Only one terminator (the end of a line) has been described in section 1.2.3. Additional character terminators may be defined by assigning to the field "end" of the charput variable controlling the reading process. If v is a charput variable, "end OF v" refers to a one-dimensional array of characters, any element of which will act as a terminator if it is encountered during the reading of a row of characters. As an illustration suppose that the following assignments are made:

```
[1:2] CHAR specialterms := "?A" ;
      end OF standin := specialterms
```

Then if c is a row-of-characters variable declared with bounds [1:4], the effect of read (c) with data

ZYXA

would be to assign ZYX to c, and reduce the upper bound of c to 3. The letter A would be the next character encountered on reading. Notice that a reference to an array must be assigned to "end OF" ; the following is illegal:

WRONG end OF v := "?A"

Any characters defined as terminators must of necessity come from the set given in section 1.2.2. An additional facility is that one other character - the horizontal tab (HT) - will always terminate a row-of-characters input, and will be translated by "read" and "get" to the space character.

The assignment

```
end OF v := NIL
```

will put the set of character terminators back to an empty array, although the end of a line or the tab character will still terminate an input. A method of overriding all terminators (including end of line and HT) without reassigning to end OF v is described in section 5.

4 FORMATS

Formatting is a technique for controlling the style of printing of output values and for recognizing fixed forms of input. Without using formats, the only means of varying the style for printing of numbers is the use of the "numberstyle" variable. Formatting provides the same facilities, and many more besides, in a compact and convenient way.

An introduction to the use of formats is given in chapters 12 and 13 of the "Algol 68-R Users Guide" (2nd edition, HMSO July 1974). This section of the manual is for reference, and describes all the available facilities, including those frames which are not described in the Users Guide.

A format is a sequence of pictures each of which, in simple cases, describes the transput of a single value. Assuming for convenience that we are dealing with output, each successive value produced by the program uses the next picture in the format as a specification of how that value is to be printed. For example, to have 3 reals printed in the style

```
1.0000
1.4142
1.7321
```

The required format would be

```
$ 1<1.4>, 1<1.4>, 1<1.4> $
```

Dollars are used as brackets for the denotation of the format in the program. The picture $1<1.4>$ says "take a new line and print the value with 1 digit before the point and 4 after". The format denotation is used as one of the parameters of a transput procedure (4.1).

This simple example enables us to introduce two important technical terms. The $<1.4>$ part of the picture is concerned only with the style used for representing the actual numerical value, and is known as the pattern for the value. The remainder of the picture (the letter 1 which takes a new line) is known as an insertion. A picture thus consists of a pattern, together with any insertions needed for layout or other purposes such as headings. If by any chance a picture should contain insertions but no pattern, the insertions are obeyed when the next pattern is dealt with. Pictures are separated by commas.

A similar format to that shown above could be used for input, though with less obvious reason. The reading of each number would be preceded by the taking of a new line, and the numbers on the data stream would cause an input fault if they did not have 1 digit before the point and 4 afterwards. Formatted input is more exacting than ordinary input. New lines and spaces before numbers are not ignored, as the format lays down exactly where the numbers are expected to appear. However, formatted input can be very useful when the data contains program steering information, as the format provides a convenient way of recognizing control messages (see 4.10.2, example 2).

Within a format denotation, the same rules about spaces and new lines apply as everywhere else in the program. Except within string quotes, spaces are ignored. New lines are also ignored.

4.1 The procedures "inf" and "outf"

The procedures "inf" and "outf" are similar to get and put, but have an extra parameter for the format. The comparison with formatless transput is shown by

<u>without formats</u>	<u>with formats</u>
read(variable-list)	inf(standin, <u>format</u> , variable-list)
print(value-list)	outf(standout, <u>format</u> , value-list)
get(v, variable-list)	inf(v, <u>format</u> , variable-list)
put(v, value-list)	outf(v, <u>format</u> , value-list)

where v is a charput variable (eg standin or standout) as described in an earlier section.

To use inf and outf in the most straightforward manner, the number of values in the variable-list or value-list should be equal to the number of pictures in the format. Each picture is "obeyed" in turn as each successive value is transput. For example, suppose that 1.00, 1.41 and 1.73 are held in an array a with bounds 1:3. Then

```
outf(standout, $ 1<1.2>, 1<1.2>, 1<1.2> $, a)
```

will print the array as

```
1.00  
1.41  
1.73
```

In this example, all 3 values are treated alike, but it is obvious that they could all have been treated differently if required.

If the number of pictures in a format is smaller than the number of values to be transput, the format is exhausted before transput is complete. The sequencing through the format then starts again at the first picture. This enables us to shorten the above example. Thus

```
outf(standout, $ 1<1.2> $, a)
```

would give exactly the same result as before, by cycling round the format 3 times over. One should always make sure that the format is left in a completed state at the end of the call of inf or outf, for if this is not done, the next call of inf or outf with that format will cause a fault and produce the error message FORMAT IN USE.

As an alternative to using inf and outf, formatted transput may be carried out with the procedures "in" and "out" described in 4.11. These procedures work to formats specified in advance, and need not use the whole of a given format in one call. Whichever procedures are used for formatted transput, ordinary unformatted transput can be used in addition and on the same channel. Calls of read, print, get or put do not interfere with the position in a format.

4.2 Standard number patterns

In Algol 68-R, simple standard patterns are provided for integers, reals and complex numbers in order to meet the commonest requirements as simply as possible. More general patterns can be constructed, character by character, as described in section 4.6. There are four forms of standard pattern; examples of these are

- <5> an integer, as space or minus sign and 5 digits
- <3.6> a real, as space or minus sign, 3 digits before and 6 after the point
- <1.4&2> a real, as space or minus sign, 1 digit before the point, 4 after and an exponent of &, plus or minus and 2 digits
- <6&2> a real with exponent but no point, ie space or minus sign and 6 digits & 2-digit signed exponent

A complex pattern is made up of a pair of real patterns separated by the letter i, which causes the symbol ? to be printed (or expected on input) in accordance with the standard convention for complex numbers (eg <1.4&2>i<1.4&2>).

The number of significant figures for output of a real should not exceed 11; further figures would be meaningless. On input, excess figures cannot be represented within the machine. The exponent for a real cannot have more than 3 digits. In all cases, leading zeros on output are replaced by spaces. Note that a real value cannot be transput with an integer pattern; this would cause the fault FORMAT MODE ERROR.

4.3 Insertions and replication

Insertions for layout can be placed before and after a standard pattern but not inside it. (To achieve that effect, see section 4.6.) The notation for insertions is

- x space
- y backspace
- l new line
- p new page
- k for setting the character position in a line (see below)

These insertions may be preceded by an integer known as a replicator. For example, 3x means three spaces, 3lx means three new lines and one space (no replicator means 1), 3l2x three new lines and two spaces. The replicator before k specifies the character position in a line, eg 5k sets the character position to the fifth in the line. Although an integer can be used for replicating an insertion, it cannot be used to replicate a pattern. Patterns cannot be replicated, but a sequence of pictures can be replicated as described in section 4.4.

/over

Insertions can also be literal character strings, which enables headings to be included as part of a format. Such literal insertions, which must be written inside string quotes, can be replicated if desired. However, adjacent strings cannot be written in the format because "" within a string denotation is always taken to mean that the quote symbol itself is wanted inside the string. (For example, "a ""mome rath"" is the denotation for A "MOME RATH".)

4.3.1 Examples of insertions

Example 1 - when are insertions obeyed?

```
FOR i TO 4 DO outf(standout, $ 1 5k <1>, 5x <1.3> $, (i, sqrt(i)))
```

The output of i begins with the new line and the setting of the character position within the line (to 5); the value of i is then printed with the pattern <1>. This completes the first picture. The output of sqrt(i) first takes 5 spaces and then prints the value with the pattern <1.3>. The actual output is

```
1      1.000
2      1.414
3      1.732
4      2.000
```

Example 2 - literal insertions

```
INT money := 1234;
outf(standout, $ 1 5k "there is £" <4> " in the bank" 6!"!$ , money)
```

gives the output

```
THERE IS £ 1234 IN THE BANK!!!!!
```

Example 3 - output of a structure

```
[1:3]STRUCT(INT age, REAL height) data;
read(data);
outf(standout, $ 10k"age"<2>, 20k"height"<1.2> 1 $, data)
```

gives the output

```
AGE 20      HEIGHT 1.85
AGE 34      HEIGHT 1.82
AGE 27      HEIGHT 1.90
```

The two pictures in this format are used three times over before the values are exhausted.

4.4 Replication of pictures

Any sequence of pictures can be replicated. Thus

4(picture, picture, picture)

replicates the 3 pictures cyclically 4 times. It can be preceded and followed by insertions and used as a composite picture in a list of further pictures. For example,

```
[1:5]INT a := (1, 2, 3, 4, 5);
outf(standout, $ 1 4(<1> "," ) y " ", "and" <1> $, a)
```

gives the output

1, 2, 3, 4 AND 5

The format in this example is equivalent to

\$ 1<1>",", <1>",", <1>",", <1>","y" ", "and"<1> \$

which shows that the insertions preceding the replication are treated as part of the first picture, and those following it as part of the last.

4.5 Dynamic replication

Replicators can be written as integer constants whenever the number of repetitions of an insertion or of a picture-list is the same each time the format is used. Occasionally it may be necessary to use a variable replicator, and this must be written in the form

n(serial clause)

The letter n is merely a symbol to indicate variable replication, and it must be followed by a bracketed variable or expression which will deliver the integral number of repetitions needed. If the expression delivers a negative value, the replicator is assumed to be zero. As an example,

```
FOR i TO 5 DO outf(standout, $ 1 n(i)k <1> $, i)
```

gives the output

1
2
3
4
5

4.6 Construction of number patterns from frames

The standard patterns for integers, reals and complex numbers (section 4.2) provide all the facilities commonly required. To gain additional flexibility, the pattern for a number can be built up from smaller units known as 'frames', suitably replicated where appropriate. One advantage of this over the use of the standard patterns is that it enables insertions to be placed within the pattern of a number. For example, to transput the integer one million without a preliminary space or sign, the pattern could be 7d, where d is the frame for a digit, but spaces could be inserted between groups of three digits by writing 1d x 3d x 3d. Completely flexible arrangements for the representation of numbers are surprisingly intricate, so where possible the standard patterns are much to be preferred.

The overall pattern for an integer is constructed from an optional sign frame and suitable digit sequences. The pattern for a real is constructed from digit sequences and frames for sign, point and exponent symbols in appropriate places. A real must have a point or an exponent symbol or both. A complex pattern is made up of an i frame between two real patterns. If no sign frame is given, no sign or space is output, whilst for input no sign or space will be expected. For negative numbers a sign frame must always be present.

The available frames for constructing number patterns are:

- + sign (as + or -), not replicatable
- sign (as space or -), not replicatable
- d digit
- u digit with zero suppression. Zero is replaced by space if it is the first digit in a number or exponent, or if the previous digit was a suppressed zero.
- v digit with partial zero suppression. Like u, but does not suppress the final digit of the v-replication.
- z digit with zero suppression. Like u, but starts suppressing zeros afresh, regardless of any previous digits. This enables zeros to be suppressed in a group of digits within one number.
- .
- e exponent symbol (&), not replicatable
- i imaginary symbol (?), not replicatable

Examples for integers from -100 by 50 to 100

+3d	-3d	-3u	-3v	-ux2v	-vxzd
-100	-100	-100	-100	-1 00	-1 0
-050	-050	- 50	- 50	- 50	-0 50
+000	000		0	0	0 0
+050	050	50	50	50	0 50
+100	100	100	100	1 00	1 0

Examples for reals from -100 by 50 to 100

+3d.d	-3v.d	-3ve-d	-d.2de+d	-ded
-100.0	-100.0	-100& 0	-1.00&+2	-1&2
-050.0	- 50.0	-500&-1	-5.00&+1	-5&1
+000.0	0.0	0& 0	0.00&+0	0&0
+050.0	50.0	500&-1	5.00&+1	5&1
+100.0	100.0	100& 0	1.00&+2	1&2

4.6.1 Placing of the sign

In all the foregoing examples, the sign occupies a fixed position in front of each number, even when the number starts with spaces in place of zeros, brought about by the use of an initial u, v or z frame. To have the sign positioned contiguous with the actual digits of the number - after the suppressed zeros - the sign frame is written after the u, v or z frame instead of earlier.

Examples for integers and reals from -100 by 50 to 100

4v+	-4v.d	4v-.d	-de+2v	-de2v+
-100	- 100.0	-100.0	-1&+ 2	-1& +2
-50	- 50.0	-50.0	-5&+ 1	-5& +1
+0	0.0	0.0	0&+ 0	0& +0
+50	50.0	50.0	5&+ 1	5& +1
+100	100.0	100.0	1&+ 2	1& +2

4.6.2 Definitions of the standard number patterns

The standard number patterns described in section 4.2 all use contiguous signs and may now be defined in terms of frames. The equivalents for the examples given in 4.2 are

<5>	5v-
<3.6>	3v-.6d
<1.4&2>	1v-.4de2v+
<6&2>	6v-e2v+

4.6.3 Left justification on output

On output only, numbers can be left justified by the use of frames wu and wv instead of the usual u and v:

- wu like u, but the spaces arising from zero suppression appear at the end of the picture (or immediately after ? if the spaces come from the first real of a complex number)
- wv like v, but the spaces arising from zero suppression appear at the end of the picture (or immediately after ? if the spaces come from the first real of a complex number)

Examples for -1000, -100, -10 and 0 with various patterns

-4wv	-4wv.d	+3wv.de+d
-1000	-1000.0	-100.0&+1
-100	-100.0	-100.0&+0
-10	-10.0	-100.0&-1
0	0.0	+0.0&+0

4.6.4 Frame suppression and omission of leading zeros

The prefix of the letter s to certain frames has the effect of suppressing these frames completely (ie without any replacement by spaces). When s is used with u or v, a facility available only for output, the effect is the complete suppression, ie omission, of leading zeros.

- se)
si) on output the character is omitted and on input nothing
s.) is read
- sd) on output the character is omitted, and on input, though
sz) nothing is actually read, transput behaves as though 0 had
been read
- su) available only on output, these frames behave like u and v
sv) but leading zeros are omitted instead of being replaced by
spaces

As an example of sv and s. in a reasonable context

```
outf(standout,  
      $ 1 "there is £" 7svs."--"2dx "in the bank" $,  
      (3.5, 1000))
```

gives the output

THERE IS £3-50 IN THE BANK
THERE IS £1000-00 IN THE BANK

4.7 Special types of number

4.7.1 Long integers

Long integers can be transput with integer patterns. Up to 14 digits can be input (within the limits set by overflow), but on output the number of digits is arbitrarily restricted to 11. For output of a long integer to maximum capacity, a radix other than 10 must be used (4.7.2).

4.7.2 Radix other than 10

Values of mode INT and LONG INT can be transput in the scales of 2, 4, 8, 16 or 64 by starting the pattern (but not any of the standard patterns of 4.2) with 2r, 4r or 8r etc. There is no artificial restriction on the length of an output LONG INT when such a radix is used. The 'digits' in base 16 are

0 1 2 3 4 5 6 7 8 9 A B C D E F

whilst in base 64, the character used for the digit n is given by REPR n. If no sign frame is given after the radix, the value is printed or input as a machine word.

4.8 Ignoring input spaces and new lines

In formatless input by "read" or "get", a numerical value can be preceded by an unknown number of spaces or new lines. The whole concept of formatting runs counter to this sort of flexibility, but there are occasions when one may wish to retain such freedom whilst exploiting the other facilities which formatting has to offer. In Algol 68-R, the need is met by the 'g' pattern, which specifies that the corresponding value be transput as though by "standput" or "get". Thus, when the pattern g is used on output, the rules of section 1.3 apply, regardless of whether any assignments have been made to the library "numberstyle" variable. For input, provided that numberstyle has not been altered by assignment, the rules of 1.2 apply, so that spaces and new lines in front of a number are simply passed over.

4.9 Booleans and characters

For booleans and characters, the following patterns are used:

- b boolean value (transput as T or F)
- a single character (mode CHAR), but see text below
- g causes transput of any value as if by standput or get.
The g pattern may therefore be used for character sequences
(modes []CHAR, STRING, BYTES, LONG BYTES, LONG LONG BYTES)
as described in sections 1.2 and 1.3.

The 'a' pattern may also be used as a frame, which means that it can be replicated for transput of character sequences. The sum of the replicators of all the 'a' frames in the pattern must be equal to the number of characters to be transput. When doing this for input, it follows that the number of characters must be known in advance, so string terminators are unnecessary, and the automatic adjustment of the upper bound, which occurs in formatless transput, does not apply.

An sa frame is also available for suppressing a CHAR. On output the character is omitted, and on input - though nothing is actually read - transput behaves as though a space had been read.

4.10 Choice patterns

Booleans and integers can, if desired, be transput as character strings chosen from a set specified in a 'choice pattern'.

4.10.1 Boolean choice pattern

To transput boolean values as arbitrary strings, the pattern

b(string, string)

is used, where the strings are written between quote symbols in the usual way. The strings become the representations of TRUE and FALSE respectively. For example,

```
BOOL female := FALSE;  
outf(standout, $ 1"there are " 2d, b("girls", "boys") " in the class" $,  
      (24, female))
```

gives the output

THERE ARE 24 BOYS IN THE CLASS

On input, the second string must not start with the first, eg

b("with", "without")

since WITH and WITHOUT on the input stream would both be matched to "with" as soon as 4 characters had been read. The answer is to reverse the order.

4.10.2 Integer choice pattern

The choice pattern

c(string, string, string, etc)

causes the strings to become the external representations of 1, 2, 3, etc respectively. An attempt to output a negative integer, zero, or an integer greater than the number of strings included in the choice pattern will lead to a NUMBER FORMAT ERROR. On input, the first string in the pattern recognized in the input stream determines the integer value read in, so the whole of any specified string must not be the start of a later one.

Example 1

To read in a sentence such as CURVE OF THE SECOND DEGREE and set an integer accordingly,

```
INT degree;
inf(standin,
    $ "curve of the " c("first", "second", "third") " degree" $, degree)
```

Example 2

The following extract of program shows how control information can be input and acted on, along with ordinary numerical data:

```
REAL x, y;
INT n;
WHILE inf(standin, $ c("end", " ", "x=", "y=", "run", "") $, n);
      n > 1
DO CASE n-1 IN
      SKIP, read(x), read(y), serial clause, read(newline) ESAC
```

Note the use of an empty string as the final item of the choice pattern. This acts as a default condition; if none of the preceding patterns has been matched by the input, the empty string is taken as the match and nothing is actually read. In this particular example, the resulting action is to take a new line. The variables x and y in the program will be set by 'commands' like

Y=0.538, (the comma and this comment will be ignored)

in the data stream. The serial clause will be obeyed by the command

RUN

and the program will be stopped by the command

END

Spaces before commands will be ignored, and so also will whole lines which do not start like commands.

4.11 The procedures in, out and format

When "inf" or "outf" is used, the required format is included in the procedure call and must be fully 'used up' in that call, for otherwise the next call upon the same format causes the error FORMAT IN USE. In some applications, particularly the laying out of tables, this may prove highly inconvenient. An alternative more flexible system is to specify the format for use with the required charput by calling the procedure

```
format(charput variable, format)
```

Thereafter, transput is effected by one or other of the procedures "in" or "out" which use the given format automatically, and if called repeatedly work through the format progressively. The manner of calling in or out is

```
in(charput variable, variable-list)  
out(charput variable, value-list)
```

When either of these is used in a loop, the call of "format" would normally be outside the loop to control transput of all the values. Indeed, the same call of "format" must not be obeyed a second time until the format is completely used up. In this respect, "format" is like "outf" (which simply calls "format" and then "out").

Example 1

To print the letters A to X with spaces between groups of three letters.

```
format(standout, $3(a)3x$);  
FOR i TO 24 DO out(standout, REPR(32+i))
```

which gives the output

```
ABC DEF GHI JKL MNO PQR STU VWX
```

It may be worth remarking that 3a in this format would have been wrong, as out is dealing with one character at a time. The compound picture 3(a)3x is equivalent to

```
a, a, a3x
```

used cyclically in this example, 8 times over.

Example 2

As above, but completing the alphabet with the group YZ and a new line.

```
format(standout, $8(3(a)3x), 2(a)1$);  
FOR i TO 26 DO out(standout, REPR(32+i))
```

4.11.1 Clear format

The necessity for using the whole format before repeating a call of inf, outf or format is designed to help the programmer by providing him with a run-time check that the format is not out of step with the values being transput. Occasionally there may be good reason for wishing to abandon the unused portion of a format before using the same format again. There is a mechanism for doing this, which is to call the procedure "clear format" before a repeat call of "format" (or inf or outf, which themselves call it). The procedure "clear format" has one parameter, the actual format to be cleared, and delivers this same format as its result. It can therefore be used as shown in the following example:

```
format(standout, clear format($p"table of results"21 50(<3>, 5x<4.1>1)$));  
FOR i TO n DO out(standout, (i, result[i]))
```

This example starts a new page with a heading and prints two columns of figures, eg

TABLE OF RESULTS

1	1234.5
2	1271.4
3	1308.4

and takes a new page after every 50 lines of results. If n is not a multiple of 50, the format is left uncompleted. However, if the given extract of program were in some larger loop, further tables could be satisfactorily output, starting each time at the beginning of the format.

It is impossible to apply the procedure "clear format" to a format which has already been set up without it, unless the format has been given an identifier as explained in 4.12. To see this, consider the following attempt:

```
format(standout, $p 50(<1.6&2>1)$);  
----  
----  
clear format($p 50(<1.6&2>1)$);
```

The last line will indeed clear the format given as its parameter, but this is a new format, and moreover one which has not been associated with any charput variable. The net effect will be nothing. The original format will be unaffected.

4.11.2 Advancing the position in a format

Normally a picture of a format is only obeyed as a value is transput. Occasionally we may wish to leave out a value. Its picture may then be passed over by calling the procedure

skip picture (charput variable)

This moves the current position in the format associated with the charput to the beginning of the next picture (possibly the end of the format), without obeying any parts of the current picture.

4.12 Identification of formats

In all previous sections, the only means used for representing a particular format has been the use of an actual format denotation, written with \$ brackets. A format can, however, be represented by an identifier. The mode of a format is FORMAT, and identity declarations may be written in the usual way, eg

```
FORMAT voltstyle = $ <3.1> "volts" $;
```

Formatted output of a voltage v (say) could then be expressed, using outf, by

```
outf(standout, voltstyle, v)
```

It is important to realize that whenever the identifier "voltstyle" is used, it means exactly the same format (not a copy of it). This fact is brought out by considering

```
FORMAT m = $ .... $;  
format(standout, m);  
----  
----  
clear format(m)
```

This clears m. The clearing would not have been achieved if the format denotation had been written in place of m in both places (discussed in 4.11.1).

The main purpose of a FORMAT declaration is to enable the same format to be used at different places in a program.

4.12.1 Formats within formats

Whilst the naming of a format (4.12) enables the same format to be used in different contexts, it does not in itself meet the frequent requirement to embody a given format at different places inside other formats. This needs the additional construction

f(serial clause delivering a format)

which can be used inside a format denotation. The f-construction cannot be replicated, but can be preceded and followed by insertions to make what looks like a simple picture. In fact, the delivered format can be a sequence of pictures. As a simple example of f,

```
FORMAT money = $ "£" 7sv "in credit" $;  
STRUCT(STRING name1, INT sum1, STRING name2, INT sum2)s;  
inf(standin, $ lg, g, lg, g $, s);  
outf(standout, $ "mr " g, " is " f(money),  
      " and mr " g, " is " f(money) " also." $, s)
```

When supplied with the input data

JONES
400
SMITHERS
3

this piece of program gave the output

MR JONES IS £400 IN CREDIT AND MR SMITHERS IS £3 IN CREDIT ALSO.

Care must be exercised in the use of "clear format" applied to a format containing embedded formats, as the embedded formats will not be cleared.

4.12.2 Multi-channel transput with formats

In Algol 68-R, a format cannot be associated with more than one charput variable at a time. If similar formats are required for use on parallel channels, separate copies are necessary. These can be made simply by writing the format denotation into the program as many times as required, or by using the procedure

```
PROC copy format = (FORMAT original, REF FORMAT copy)
```

For example,

```
FORMAT f1 = $ ... $;  
FORMAT f2, f3;  
copy format(f1, f2);  
copy format(f1, f3)
```

provides three independent copies of the format. The copying procedure should be called immediately after the declaration of the format variable which is to refer to the copy. It should be noted that the assignment $f2 := f1$ does not make a copy, but merely causes $f2$ to refer to the original.

4.13 Index of formatting symbols

<u>symbol</u>	<u>mnemonic</u>	<u>section</u>
.	point	4.6
+	sign	4.6
-	sign	4.6
a	character	4.9
b	boolean	4.9, 4.10.1
c	choice	4.10.2
d	digit	4.6
e	exponent	4.6
f	function	4.12.1
g	get or put	4.8, 4.9
i	imaginary	4.6
k	char position	4.3
l	line	4.3
n	dynamic rep	4.5
p	page	4.3
r	radix	4.7.2
s	suppression	4.6.4, 4.9
u	digit	4.6
v	digit	4.6
w	left justify	4.6.3
x	space	4.3
y	backspace	4.3
z	digit	4.6

4.14 Glossary of formatting terms

Format	A sequence of pictures in \$ brackets.
Frame	An atomic component of a pattern. Most frames can have replicators.
Insertion	Material included in a picture, such as new lines, to be associated with the transput of a value. Insertions can have replicators.
Pattern	Description of the style of a value, excluding any insertions. Patterns cannot have replicators.
Picture	The pattern for a value, including any insertions.
Replicator	An integer (or equivalent) prefixing an insertion, frame or bracketed sequence of pictures.
Standard pattern	A stereotyped pattern for a number, within which there can be no insertions.

O

O

O

O

5 EVENTS

The standard input/output procedures respond to certain occurrences in a way which may be inconvenient in some particular application. For example, if a number cannot be printed in the style requested, the standard response is to print the number in full and then to fault with the display NUMBER FORMAT ERROR. Although such fault stops are designed to help detect unsuspected logical errors of programming, it may on occasion seem desirable to build a different response into the program. A number format error is only one example of what is described as an 'event'. Not all events are faults; for example, every occurrence of a new line, whether on input or output, is an event. The purpose of this section is to explain the internal mechanism used by the reading and printing procedures for dealing with events, so that users with special requirements can alter the standard response.

5.1 The event field of a charput variable

When an event occurs on a particular channel, the library procedure dealing with the input or output obeys the field "event" of the charput variable controlling the channel. The event field holds a procedure of mode PROC(INT)INT, which takes one integer parameter and delivers an integer result. The library routine obeys this procedure by supplying as its actual parameter an integer denoting which particular event has occurred. To take the previous examples, if a number to be printed is too big for the current numberstyle setting, "print" will call

(event OF standout)(-9)

while if a program obeys the statement

newline(in1)

where "in1" is a charput variable controlling an extra input channel, the newline procedure first calls

(event OF in1)(-1)

For convenience, the above events will be referred to as event -9 and event -1 respectively. The event procedure for each charput variable is normally the routine

(INT eventnumber) INT: BEGIN -1 END

so that the effect of these calls is always to deliver a negative result. By convention, this indicates to the input or output procedure which encountered the event that the standard response to that event is required. Had a positive result been delivered, an alternative non-standard response would be obtained; these alternative responses are listed in section 5.4. As an example, delivering a positive result to event -9 causes the offending number to be omitted from the output and the program to continue with no fault indication. Bypassing of the standard response may not in itself be a constructive alteration, but if a new event procedure is being written it can embody any desired action, as described in 5.2.

5.2 Changing the event field

The event field of any charput variable can be changed. If v is a charput variable, "event OF v" is a reference to a procedure of mode PROC(INT)INT, and a different procedure (of the same mode) may be assigned to it. A call such as

(event OF v)(-2)

from an input/output routine will then obey whatever procedure is currently assigned to event OF v. Such a procedure can determine which event has occurred by testing the value of the parameter passed to it, and for particular events deliver a non-negative result to change the response of the input/output to those events. Further, since the body of the procedure will be obeyed whenever an event happens, extra actions can be written into it to be performed at certain event points. For example, if the parameter is -2 (meaning that "newpage" has been called), a page number could be printed. To ensure the normal input/output action in the case of those events not being altered, the procedure must deliver a negative result for all other values of its parameter.

The following outline shows the form of an assignment to an event field.

```
event OF v :=  
  (INT eventnumber) INT:  
    BEGIN  
      IF eventnumber = an event the program wants to alter  
      THEN extra action if wanted;  
      ----;  
      ----;  
      IF input or output to behave in the standard way  
      THEN -1 ELSE 0 {to by-pass the standard response}  
      FI  
      ELSE -1 {normal response for all other events}  
      FI  
    END ;
```

The assignment must be made after the call of "openc" for the charput variable v, as openc initialises all the fields of a charput variable to their standard values.

5.3 Examples of event procedures

These examples show some common uses of event procedures. For simplicity each procedure considers only one event, and delivers the normal (negative) result for all other values of its parameter. A full list of event numbers and the circumstances in which they occur can be found in section 5.4.

5.3.1 Ending input when the end of a file is reached

When an extra input channel has been opened in a program as a "file reader" an attempt to read beyond the end of the George file which has been assigned to the channel can be detected by the input procedures. The event field of the charput variable controlling the channel (say in1) is then obeyed with -12 as its actual parameter. The following procedure jumps to the label "all in" when this occurs.

```
event OF in1 := (INT eventnumber) INT:  
(IF eventnumber = -12 THEN GOTO all in ELSE -1 FI);
```

The normal input channel (channel 0) used by "read" is not a "file reader", and reading beyond the end of a file assigned to it will give a George error message. The charput variable standin used by "read" can however be associated with a file reader channel (with channel number 1 say) by the statement

```
openc (standin, file reader, 1);
```

Data for "read" will then be taken from the George file assigned to *fr1 in the job description and event -12 will occur when the end of this file is reached.

5.3.2 Continuing after a number format error on output

When an output procedure detects that a number to be output is too large for the requested style, it obeys the event procedure with parameter -9 and if the result delivered is negative outputs the number in default style and calls event -10. If the result from -10 is negative, the program will fault. A program could replace the default printing with some other message, and then continue as though there had been no error, for example

```
event OF standout := (INT eventnumber) INT:  
(IF eventnumber = -9 THEN print("*****"); 0 ELSE -1 FI);
```

Alternatively, by altering event -10, the program could be made to continue after the default printing:

```
event OF standout := (INT eventnumber) INT:  
(IF eventnumber = -10 THEN 0 ELSE -1 FI);
```

5.3.3 Ending input of an indefinite sequence of numbers

Suppose that a program is to continue reading numbers until a terminating character (say "/") is encountered. Any number must start with a sign or a digit, so that a character "/" after the last number will be recognized as illegal by the procedure "read". In this case event OF standin is called with a non-negative parameter. The following example detects this and inserts the action of jumping out of the input to the label "all in" set in the program. Note that it distinguishes the deliberate slash from an error caused by a mis-typing of the data (eg a comma for a decimal point!), and still faults ILLEGAL CHARACTER in the latter case. This is done by backspacing over the illegal character and re-reading it to see if it was a "/".

```
BEGIN
    REAL x;
    event OF standin := (INT eventnumber)INT:
        (IF eventnumber >= 0
        THEN IF CHAR a;
            read((backspace, a));
            a = "/"
        THEN GOTO all in
        FI
    FI;
    -1);
    ----;
    DO (read(x); ----);
all in:
    ----
END
```

5.3.4 Reading rows of characters

The full rules for reading rows of characters (events -4, -5, -6 and 16) are given in 5.4.2. This section gives two simple examples.

When an ordinary []CHAR variable (ie not a STRING) is read, and there are not enough characters on the current line to fill the array, reading stops at the end of the line and the upper bound of the array is reduced to suit. The following event procedure deals with the end of the line itself in these circumstances, so that reading will carry on (over several lines if necessary) until the array is filled.

```
event OF standin := (INT event number) INT:  
  (IF event number = -5 THEN newline(standin); 0  
   ELSE -1  
   FI);
```

A more common requirement is to stop reading characters when the end of the line is reached, and then, instead of reducing the upper bound of the array, fill up the rest of the array with space characters. This must be thought of in two stages. The first stage is to ensure that the upper bound of the array remains fixed. This is done by delivering a non-negative result in response to event -4, which will work for ordinary []CHAR variables though not for STRING. Assuming this has been done, event 16 will occur if the end of a line is reached before enough characters have been read. (This same event 16 occurs if the end of a line is reached during input of a BYTES, LONG BYTES or LONG LONG BYTES.) This event must be intercepted, and ABS " " delivered as its result, to introduce a space character. Event 16 will occur again if there is still room in the array, until enough extra characters have been provided. Notice that the reading position remains at the end of the line, so newline must be called before attempting to read another []CHAR.

```
event OF standin := (INT event number) INT:  
  (IF event number = -4 THEN 0  
   ELSF event number = 16 THEN ABS " "  
   ELSE -1  
   FI);
```

5.3.5 The new line and new page events

Whenever a new line is required on an input or output channel controlled by a charput variable v, the procedure newline(v) is obeyed. This procedure may have been called explicitly in a program or implicitly by "put" say, when for example a number to be printed will not fit on the current line. At every call of "newline" the procedure (event OF v)(-1) is obeyed first. If this procedure delivers the normal negative result a further procedure, named "nextline", is automatically called and actually performs the input or output of the new line. New pages are treated similarly; at an explicit call of newpage(v) or an implicit call (see below for when this occurs) the procedure (event OF v)(-2) is called. If this delivers a negative result the procedure "nextpage" takes the new page. If either event procedure delivers a non-negative result, input or output continues without calling nextline or nextpage.

This mechanism gives two main ways of altering the new line and new page events. Extra actions can be inserted in the event procedure and a negative result delivered; these actions will then be performed before the new line or page is taken. Alternatively the appropriate procedure "nextline" or "nextpage" can be called in the event procedure, followed by actions to be taken at the start of the new line or page, and a non-negative result delivered.

The procedure "newline" itself should not be called inside an event procedure when the event number is -1 as this would lead to a program loop, with (event OF v)(-1) being called continuously (and recursively). Similarly "newpage" should not be called when the event number is -2. The more primitive procedures "nextline" and "nextpage" are designed to overcome this difficulty (they do not call the "event" procedure) and are for use only in these circumstances. They must always be called if the event procedure delivers a non-negative result, to ensure that the new line or page is actually taken.

The following example shows an event procedure which inserts page headings on the channel controlled by the charput variable v. In addition each line of output (except for the page heading) is indented by 6 spaces. (For another version of this example, using formatted transput, see 5.5.2.) Implicit calls of newpage, and hence the event procedure, occur at the initial transput on a channel (see section 5.3.5.1), at the end of a program for the charput variables standin and standout, at the closing of a channel by "closec" and (if pagesize for the channel has been set) when the number of lines printed is equal to pagesize.

```
event OF v  :=  (INT eventnumber) INT:  
BEGIN      IF eventnumber = -1 THEN  
            nextline (v);  
            TO 6 DO space(v);  
            0  
        ELSF eventnumber = -2 THEN  
            nextpage(v);  
            put(v, ("PAGE", pagenumber(v)));  
            newline(v);  
            0  
        ELSE -1  
        FI  
END;
```

5.3.5.1 The initial new page event; use of "line number" and "page number"

When a channel is newly opened, the procedures "char number", "line number" and "page number" all deliver zero, and the first transput on the channel is preceded by a call of event -2 to get onto line 1 of page 1. This arises because a call of nextline is automatically inserted if the first transput is not an explicit call of "new line" (leading to nextline) or "new page". The procedure nextline always checks to see if the line number is zero, and if it is calls new page and hence event -2. If a string reader or similar channel is being used, the initial event -2 means that nextstring need never be called outside an event procedure, provided that it is called on events -1 and -2. If nextstring is first called in the main body of a program, any assignment of an event procedure to detect the initial event -2 must be made before the call of nextstring.

The line number of a channel is incremented by nextline and the page number by nextpage, which also sets the line number to 1, so the values delivered by page number and line number during events -1 and -2 depend on whether or not these primitive procedures have been called. (They are called after the event in the usual case.) If "page size" (described in section 3.2) has been set, each call of nextline increases the line number first and then calls newpage if it is greater than pagesize. Thus the line number at the beginning of event -2 will be greater than pagesize in this one circumstance, and this can be used to distinguish between automatic and direct program calls of newpage (for example, to give an abbreviated continuation heading or a full page heading).

5.4 Table of event numbers

The table overleaf lists all the event numbers, indicates whether they are for input or output, gives a brief definition of the circumstances in which they occur, and (on the right hand page) states the actions taken by the input or output procedures after a negative or a non-negative reply. References to other sections are given inside square brackets.

Note that the response to a non-negative reply often assumes that the situation causing the event has been dealt with before the reply is given; input or output will continue as if no event had occurred. In those responses marked with an asterisk (*), failure to take care of the situation will cause a program spin, that event being called repeatedly.

LIBRARY

Input/Output

5.4 contd

event number		meaning
≥ 0	I	Unexpected character encountered, or end of line during fixed length input of characters [5.4.1]
-1	I 0	A call of "newline" [5.3.5]
-2	I 0	A call of "newpage" [5.3.5]
-3	I	The next character to be input will be the first of a new page
-4	I	[5.4.2] Called before input to a non-flexible row of characters except when controlled by format a-frames
	0	Called before output of a row of characters, or BYTES, etc
-5	I	Called during variable length input of characters [5.4.2] when end of line is met
-6	I	[5.4.2] Called during variable length input to a row of characters when the HT character or a character in "end OF charput" is met
-7	I 0	End of the format
-8	0	Mode of item does not match format (output only). This, and event -13, includes the case where the number of characters in a row does not correspond with the controlling a-frames
-9	0	Item does not fit the required style (output only)
-10	I	Item does not fit the required style (input)
	0	Only called directly from event -9 (output)
-11	I 0	Attempt to transput outside the line limits
-12	I	End of the input file (file reader channels only)
-13	I	Mode of item does not match format (input)
	0	Only called directly from event -8 (output)
-14		Requested format is already controlling transput

event number	action on negative reply	action on non-negative reply
≥ 0	Fault display ILLEGAL CHARACTER	See 5.4.1
- 1	Calls primitive procedure "nextline"	Continues without calling nextline*
- 2	Calls primitive procedure "nextpage"	Continues without calling nextpage*
- 3	Continues normally	Continues normally
- 4	Reading of this row will be variable length input of characters [5.4.2]. Continues (event -11 will be called during output if there is not enough room on the line)	Reading of this row will be fixed length input of characters [5.4.2]. A new line will be taken before output if there is not enough room for the whole row
- 5	Variable length input is terminated	Input to the row continues; the new line must be dealt with by the user*
- 6	Variable length input is terminated	Input to the row continues; the terminator must be dealt with by the user*
- 7	Restarts format from the beginning	Continues; the user must provide a new format*
- 8	The item is output in default style, and then event -13 is called	The current item is skipped [5.4.3], output continues with the next one
- 9	The item is output in default style, and then event -10 is called	The current item is skipped [5.4.3], output continues with the next one
-10	Fault display NUMBER FORMAT ERROR	The current item is skipped [5.4.3], transput continues with the next one
-11	Fault display END OF LINE	Continues*
-12	Fault display END OF FILE	Continues; more characters must be provided by ASSIGNing another file and then calling newline
-13	Fault display FORMAT MODE ERROR	The current item is skipped [5.4.3], transput continues with the next one
-14	Fault display FORMAT IN USE	The format is cleared and associated with the charput variable

*Program spin may be caused if the situation is not dealt with correctly

5.4.1 Non-negative event numbers

The event procedure is called with a non-negative actual parameter whenever an unexpected character is encountered, or when end of line is reached during fixed length character reading (see 5.4.2 for definition of 'fixed length' reading). The event number is the integer value (ABS) of one of the characters which would be correct; the particular one which is taken is shown below:-

ABS "0"	(0) if a digit is expected (Note 1)
ABS "?"	(15) if ? of a complex number is expected
ABS " "	(16) if a fixed length reading of characters is still in progress at the end of a line
ABS "+"	(27) if a sign is expected (Notes 1, 2)
ABS "T"	(52) if T or F is expected in unformatted input
ABS symbol	in formatted input for the appropriate symbol when a point-symbol or exponent-symbol is expected, or a character from a literal

Note 1 In unformatted input, if an unexpected character is encountered at the start of an integer, real, or complex number, either event number 0 or 27 may occur.

Note 2 In formatted input of numbers, event 27 will occur if a sign frame is specified but the corresponding sign (or space) is not encountered. If the sign frame is "-", 29 (ABS "-") would be correct, not 27.

If a negative reply is given, the fault display ILLEGAL CHARACTER occurs. If a non-negative integer is returned, the character it represents is taken as the next input character, thus replacing the illegal (or non-existent) one. The event number itself is suitable for the reply, except to event 27 (see note 2). Other integers in the range 0-63 may be delivered in the case of event 16. When a number is about to be read (events 0, 27 and some of the ABS symbol cases for formatted input) the item will be skipped (see 5.4.3) if an inappropriate integer (say 64) is delivered.

5.4.2 Fixed and variable length input of characters

The input of certain items involves the reading of a number of characters which may either be known at the beginning of the item, or may be determined by the characters encountered. This gives two different types of input of characters with different associated events - 'fixed length' input and 'variable length' input.

Fixed length input of characters occurs when the exact number of characters which will constitute the item is determined before input occurs. This is the case for input to BYTES, LONG BYTES, and LONG LONG BYTES variables, to non-flexible []CHAR variables provided a non-negative reply was given to the preceding event -4, and to character variables or flexible or non-flexible []CHAR variables controlled by format a-frames. If the end of line is reached before sufficient characters have been encountered, event 16 occurs.

Variable length input of characters occurs when the number of characters is determined dynamically. This is the case for unformatted input to non-flexible []CHAR variables (provided the standard negative reply was given to the preceding event -4) and to flexible []CHAR variables. The input is terminated by any of the following occurrences: the row is non-flexible and is filled, a character in the field "end" of the charput is encountered (see 3.5), the HT character is encountered, or the end of the line is reached. Event 16 is not called. The terminators can be overridden by returning non-negative replies to events -5 and -6 (see example in 5.3.4).

5.4.3 Skipping an item

The action taken on receiving a non-negative response to certain events is to skip the current item in the list of items being input or output. With the exception of complex numbers, this action is defined as follows. There is no input to (or output of) the current item; an item is one field of a structure, one element of an array, or one simple variable, in the list of variables to be input or output. At the same time the format (if any) is kept in step by the current picture being skipped, no insertions being performed after detection of the error.

Complex numbers may cause some difficulty. If a number format error (events -9 and -10) occurs during output of the real part of a complex number, the skipping will be to the imaginary part, with the format (if any) being skipped up to the i-symbol. Otherwise skipping is to the next item after the complex number. Note that if an error occurs during the imaginary part, the real part will already have been transput.

5.5 Writing general event procedures

Section 5.3 describes some simple event procedures, each one altering the response to a particular event, and taking the normal system action in all other cases. Some of these procedures could be combined to alter the response to several events simultaneously. A further common requirement is for different event settings to apply in different parts of a program. If this is the case, an extra procedure identifier, "old event" say, should be declared to keep the current event setting:

```
PROC(INT)INT old event = event OF v ;
```

A further event procedure can now be assigned, for example

```
event OF v :=  
  (INT eventnumber) INT:  
  BEGIN  
    IF eventnumber = an event to be altered now  
    THEN extra action if wanted;  
    ----;  
    IF normal response to this event  
    THEN -1 ELSE 0  
    FI  
  ELSE old event (eventnumber) {all other cases}  
  FI  
END;
```

Notice that the events not altered by this procedure are treated by calling "old event", which will be the procedure last assigned to "event OF v". This ensures that any responses altered in other parts of the program still apply. The event field can be reset at any time by

```
event OF v := old event ;
```

or, if the normal response is to be restored for all events,

```
event OF v := (INT eventnumber) INT: BEGIN -1 END;
```

5.5.1 Warning about scopes

A procedure assigned to "event OF v" may be obeyed at any time during the input or output which uses v. It is thus essential to make sure that every variable, identifier or label used in this procedure is in scope whenever input/output is taking place. Particular care is necessary when assigning to "event" inside a procedure. It is always safe if the event procedure uses only objects declared globally (ie at the outermost level of the program).

Output of a new line takes place automatically at the end of every program on the channel controlled by "standout", so care should be taken that "event OF standout" is in scope at this point.

5.5.2 Input or output within an event procedure

Extra input or output can be carried out inside an event procedure, either on a different channel or on the same one, but care must be taken if formatted transput is in use. The event procedure must not interrupt the current progression of the main program through a format by attaching a different one for its own use to the same charput variable. Instead, a new local charput variable should be declared inside the procedure and used to control the extra transput, as shown in the following example which inserts formatted page headings. The assignment of a charput variable ($w := v$ in this example) is explained in section 7.1.

```
event OF v := (INT eventnumber) INT:  
  BEGIN  IF  eventnumber = -1 THEN  
    nextline(v);  
    TO 6 DO space(v);  
    0  
  ELSF eventnumber = -2 THEN  
    CHARPUT w:= v;  
    nextpage(w);  
    outf (w, $ "PAGE" 3u 1 $, pagenumber(v));  
    0  
  ELSE -1  
  FI  
END;
```

The library procedure

```
PROC current charput = REF CHARPUT
```

delivers a reference to the charput variable controlling the current input or output operations. This can be used to write a charput-independent event procedure which could then be assigned to the event field of several different charput variables.

6 PRIMITIVE INPUT-OUTPUT PROCEDURES AND HANDLING NON-STANDARD CHARACTER SETS

Input and output have been described so far in terms of the 64 characters of mode CHAR, corresponding to the ICL graphic set. However a normal file will contain characters from the larger set of 128 available with 8-track paper tape, and this section describes procedures for dealing with this larger set both on input and output, and including the case of non-standard codes.

Input to a program usually takes place as a result of obeying the library procedures read, get or inf, which take successive characters from a file and generate the value required. For example, they must do the multiplications necessary to convert the string of characters 1.23&4 to a binary floating point number. Rather than fetch characters directly, read and get etc call a primitive procedure which is kept in the field in OF the charput variable controlling the transput. This delivers an integer in the range 0-127 which represents the character read (taking account of shifts) according to the table in section 6.5. The full character set is then condensed into the graphic set: lower case letters are converted to upper case, control characters such as CR, LF are ignored and {|} are changed to @[\$]↑ respectively. If it is important to know which characters in the full set have been read, the user must call in OF charput variable directly, as described in 6.1.

The charput variable is initialised by calling the procedure openc, which will set the in field to the library procedure "standard in". A different procedure may subsequently be assigned to this field so that characters may be translated before passing them on. This makes it possible to do code conversion without having to do number assembly as well. Writing suitable replacement procedures is described in section 6.2.

An analogous situation exists for output. The procedures put, print etc call the procedure kept in the field out OF charput variable to generate characters, indicating which one is required by means of an integer parameter. Only graphic set characters will be generated in this way, so to output characters in the full set the procedure must be called directly. As with input the charput variable is initialised on opening, in this case to the procedure "standard out", and may be subsequently overwritten with a different procedure for the purposes of code conversion. This is described in sections 6.3 and 6.4.

6.1 Calling the procedure "in OF"

The basic input procedure held in the field "in" of a charput variable, v say, is called by writing

in OF selectcharput(v)

Do not simply call "in OF v". At this level, transput takes place using a current charput, and this must be correct when the procedure is obeyed. The library procedure "select charput" ensures this by changing the current charput to its parameter, and delivering this as its result.

The standard procedure held in "in OF" delivers an integer, in the range 0-127, representing the character read according to the table given in section 6.5. When the end of a line is reached, it delivers a negative value, and the procedure new line must then be called. Note that for a file reader channel, George will never pass line feed (LF) characters to the program.

6.2 Changing the "in OF" procedure

By assigning a procedure to the field in OF a charput variable it is possible to translate characters, ignore the ends of lines or do special actions whenever certain characters are input, without losing the facilities of "get". The new procedure will need to use "standard in" for actual inputting and then transform the integer it delivers to the required code from the table in 6.5. Standard in should only be called in this way; it should never be used outside a procedure being assigned to in OF.

The following example shows the assignment of a procedure for simple code transformation.

```
in OF v := INT:  
    BEGIN INT i := standard in;  
        IF i = 95 THEN 16  
        ELSF ... THEN  
            ...  
        ELSE i  
        FI  
    END
```

The assignment should be placed after the call of openc for the charput variable, and subsequent calls of get will then use the translated characters. The effect of "IF i = 95 THEN 16", which is included merely as an example, is to interpret the row of tape consisting of all holes as a representation of space instead of DEL (see table in 6.5). Other non-standard representations can be incorporated similarly as indicated by the gaps in the program. (The standard tape code is shown in GEORGE 5.4.3.) If there is a large number of changes, possibly the whole character set, it may be preferable to use an array of integers as a look-up table. Note that negative values delivered by standard in, and untranslated integers, are simply delivered as the result of the new procedure.

For a tape reader channel, "in OF" must generally be changed to ignore the ends of lines, as these will occur at arbitrary points and must be hidden from "get". This is done by the following procedure assignment:

```
in OF v := INT:  
    { INT i;  
        WHILE (i := standard in) < 0 DO newline (v);  
        i }
```

The effect of the DO-clause is that in OF v will never deliver a negative value, and consequently it will appear to read or get that there are no new lines. The procedure could of course also include a code transformation as shown above.

Any action may be done inside the "in OF" procedure, but if there is any input or output involving another channel then the original charput should be reselected before leaving (by calling select charput). By calling standard in twice, or not calling it every time, it is possible to have more or less characters delivered to get than are in the file. This will alter the action of backspace and set charnumber (also format letter y and letter k etc) whose effect is determined by the current character position *within the file*.

6.3 Calling the procedure "out OF"

The procedure held in the field "out" of a charput variable can be called to output characters not in the graphic set. The channel controlled by the charput variable should be opened as a "file punch" or "tape punch" (see section 2.5) so that shift characters are inserted automatically. The procedure takes one integer parameter representing the character required according to the table in 6.5, and delivers an integer which can be ignored. It must be called by writing

```
(out OF select charput (v))(rep)
```

to ensure that the correct charput is selected when the procedure is obeyed. Negative values of the parameter are used to call for the effect of a new line or new page, but rather than use this mechanism, call newline (v) or newpage (v) directly.

When using "out OF" it is essential to make sure that there is room on the current line for the next character (including a shift character if necessary). This can be done by counting (there is a maximum of 128 6-bit characters per line) or, for a tape punch channel the event -11 (end of line) can be trapped as shown below. The example shows a common use of out OF, to insert carriage return, line feed and 3 blanks on each call of newline or newpage so that paper tape can be produced for machines which require these characters.

```
CHARPUT v;
openc (v, tape punch, 1);
event OF v := (INT e) INT:
  (IF e = -1 OR e = -2           {new line, new page}
   THEN
     (out OF selectcharput (v))(125);
     (out OF selectcharput (v))(122); {CR LF}
     next line (v);
     TO 3 DO (out OF selectcharput (v))(112); {NUL}
     O {No default action required}
   ELSF e = -11 {line overflow}
   THEN next line (v);
     {This is an arbitrary new line ie not associated with
      a call of newline. It will be removed from the file
      because it is a tape punch channel, see 2.5.4}
     O
   ELSE -1
   FI
);
```

If an integer > 127 is presented to out OF the event procedure associated with the charput variable will be called with parameter equal to the illegal value. If the event procedure returns a positive or zero result the program will continue without outputting anything, otherwise the fault ILLEGAL CHARACTER occurs.

6.4 Changing the "out OF" procedure

By assigning a procedure to the field out OF a charput variable it is possible to translate characters as they are passed from put, outf etc, or do special actions whenever certain characters are to be output. When translating, the procedure should test the value of its parameter and call standard out with the interface value corresponding to the pattern of holes required (see table of standard tape codes in GEORGE 5.4.3). Negative values should be passed on to standard out directly. The integer value delivered by the procedure should be the same as that delivered by standard out if it was called and zero otherwise. Standard out should only be called in this way; it should never be used outside a procedure being assigned to out OF.

The following example of code transformation is the converse of the example in section 6.2.

```
out OF v := (INT i) INT:  
  (standard out (IF i = 16 THEN 95  
                 ELSF ... THEN ...  
                 ...  
                 ELSE i  
                 FI)  
)
```

This assignment should be placed after the charput variable is opened and subsequent calls of put will then generate the translated characters. The example translates spaces ($i = 16$) into the tape row consisting of all holes. Other non-standard representations may be added in a similar way with the possibility, if the number of changes is large, of using an array of integers as a lookup table.

Any action may be done inside the "out OF" procedure, but if there is any input or output involving another channel within the procedure then the original charput should be reselected before leaving. Also any translation which is not 1:1 may change the action of the positional input-output procedures.

6.5 Table of characters and interface codes

In this table the characters are in the order of their interface code, within the range 0-127.

0	0	1	2	3	4	5	6	7	8	9
10	:	;	<	=	>	?	sp	!	"	#
20	£	%	&	'	()	*	+	,	-
30	.	/	@	A	B	C	D	E	F	G
40	H	I	J	K	L	M	N	O	P	Q
50	R	S	T	U	V	W	X	Y	Z	[
60	\$]	↑	←	—	a	b	c	d	e
70	f	g	h	i	j	k	l	m	n	o
80	p	q	r	s	t	u	v	w	x	y
90	z	{		}	—	DEL	DLE	RD	PE	RD
100	PE	NAK	SYN	(ETB)	CAN	(EM)	(SS)	ACC	(FS)	(GS)
110	(RS)	(US)	NUL	SOH	STX	ETX	EOT	(ENQ)	ACK	BEL
120	BS	HT	LF	VT	FF	CR	PK	PR		

The tape code for a character in brackets cannot be produced directly from an Olivetti keyboard.

7 ADVANCED TECHNIQUES

This chapter is intended for the adventurous programmer with out-of-the-ordinary problems, who needs a deeper understanding of the structure of Algol 68-R character transput. No new facilities are introduced, and familiarity with the previous chapters is assumed. The existing facilities are combined in ways which have been found to constitute useful additional techniques.

7.1 Assigning charput variables

A charput variable contains a number of fields known to the user - "end" for input string termination, "event" for actions inserted at various moments in transput, "in" and "out" to modify the interpretation of characters, and (less obviously) a field holding the format currently in use (if any). These fields are initialised by openc and in addition, the charput is made to refer to a second structure (a 'device' structure) which varies with the second parameter of openc, and holds information about line number, page number and the type of channel (filereader, string printer etc). If another charput variable w, say is declared and assigned to in the following sequence

```
CHARPUT v, w;  
openc (v, filereader, 1);  
w := v
```

then w will contain copies of the same fields "end", "event" etc as v, but will refer to exactly the same device (note that w is *not* opened). Some of the fields of w could then be altered in the usual way with the result that transput using v or w would take place on the same channel, but yet have different string terminators, character interpretations, events or different formats. As an example, the format "one page" shown below (which can be attached to any charput variable) prints a page heading via "outf" without affecting the original charput variable.

```
PROC page title = INT:  
    BEGIN CHARPUT w := current charput;  
        outf (w, $ "page" 3v $, pagenumber(w));  
        2  
    END;  
  
FORMAT one page = $ n(page title) 1.....p $;
```

The same sort of assignment can be used to allow a program to use "print" and "read" throughout, but still transput on several different channels. If two different charput variables, say v and w, are already declared and opened on appropriate channels (say for output), then after the assignment

```
standout := v  
"print" will behave like "put(v, ...)", while after the assignment  
standout := w  
it will behave like "put(w, ...)".
```

7.2 Repositioning and reading or writing within any line of input or output

This is straightforward for input and output where there is no possibility of shift characters occurring, that is, for the standard output channel, other output channels opened as file printer or wide printer, and input channels opened as file reader or wide reader and attached to a graphic file. The repositioning procedures setcharnumber, space and backspace are described in section 3.1. In addition the current line of either input or output can always be both read and written. For example,

```
CHAR c;  
get(standout,c)
```

is possible, provided that "setcharnumber", say, has been used to set the current character position so that it is inside that part of the line which has already been written.

Care must be taken if shift characters are present, for example when reading on the standard channel (see below) or when reading a normal file. The repositioning procedures work in terms of the full character set. They take account of shifts and regard a 'character' as one of those with internal representation in the range 0-127, shown in the table in section 6.5. It is always safe to read and re-read a line, but trying to overwrite parts of a line which includes shifts may corrupt the rest of the line beyond the writing point.

The standard input channel(tr0) is peculiar. If it is attached to a graphic file, each line of input is translated by George into the shift representation (the characters \$] ↑ and ← are converted to their shift form) and a line feed character, printed as ↑*, is added at the end of each line. This must be borne in mind, as line feed is counted as a character by space, backspace and setcharnumber as described above. To avoid the problem altogether, the charput variable "standin" used by "read" can be associated with a file reader channel (numbered 1 say), instead of tr0, by the statement

```
openc(standin, file reader, 1)
```

Data for "read" is then taken from the file assigned to *fr1 in the job description and reading the graphic file is straightforward.

7.3 Two level transput

Many effects can be obtained by interposing a string printer or string reader channel (working to or from []CHAR variables) between the transput calls in a program and its final input or output. For example, it would often be convenient to print on a line printer page going down successive columns, with the width of each column determining the line length for the automatic newline facilities. The example opposite allows this by storing a complete page of characters to be output at once. The page is filled up by the program using the charput variable "column printer" for output, and printed by calls of put(line printer, ...) within the event procedure of column printer. Output must end with a call of "new page". The example relies on the use of "line number" and "page number" in events -1 and -2 as described in section 5.3.5.1.

```

CHARPUT column printer; openc (column printer, string printer, 0);
CHARPUT line printer; openc (line printer, wide printer (160), 1);

[1:60, 1:160] CHAR page; CLEAR page;

INT nr cols = 4 {or the number of columns required};
INT col width = 165 '/' nr cols - 5; {5 spaces between columns}

[1 : nr cols + 1] INT l, u;
l[1] := l[nr cols + 1] := 1;
u[1] := u[nr cols + 1] := col width;
FOR i FROM 2 TO nr cols DO
( l[i] := l[i-1] + col width + 5;
u[i] := u[i-1] + col width + 5 );

page size (column printer) := 60 * nr cols;
page size (line printer) := 60;

event OF column printer := (INT e) INT :
( IF e = -1 {newline}
  THEN INT line nr, col nr;
    line nr := 1 +
      ((col nr := 60 + line number (column printer)) '/:= 60),
    next string (column printer,
                  page [line nr, l[col nr] : u[col nr]]),
    0

  ELSF e = -2 {newpage}
  THEN IF page number (column printer) = 0 {initial page}
    THEN next string (column printer, page [1, l[1] : u[1]]),
    -1
  ELSF line number (column printer) <= page size (column printer)
    {user call of newpage}
  THEN TO page size (column printer)
    - line number (column printer) + 1
    DO newline (column printer);
    0
  ELSE {automatic newpage}
    FOR i TO 60 DO
      put (line printer, (page [i, ], newline));
    -1

  FI

  ELSE {any other event} -1
FI );

```


INTERPOLATION AND DIFFERENTIATION

Interpolation is the process of estimating the value of a function between points where the function is known. The method is used when it is impracticable or uneconomical to compute the function ab initio for any required argument. If two values are given, the function is assumed to be linear for purposes of interpolation. If three are given, a quadratic can be fitted, and so on. When the "true" function is known to be mathematically smooth, and the errors in the given values are negligible, the higher the order of polynomial fitted, the more accurate is the result obtained. However, if there are appreciable errors in the given values, the use of a high-order polynomial is not to be recommended. The interpolation process will assume that the given values are exact, and will fit a curve which is mathematically smooth but which may be highly artificial. For most practical purposes, the use of 3 or 4 points should be adequate. When in doubt, it is always advisable to carry out trial runs.

It should be remembered, particularly when using a high-order fit, that better results are obtained near the middle of the range over which function values are given, rather than near one end or beyond (extrapolation).

Numerical differentiation at an intermediate point is a process closely akin to interpolation, and the procedures equidif and interdif deliver estimates of both the function value and its derivative.

When it is required to perform interpolation in many different arrays for the same value of the argument, repeated calls of the interpolation procedure are wasteful, as the same weights for the function values are calculated every time. The weights used by the library procedures can be found by supplying the procedures with suitable groups of zeros and ones, and Lagrangian interpolation can then be programmed independently.

Inverse interpolation is the process of finding the argument at which a tabulated function is estimated to have a given value. The procedure inverlate expresses this task as that of solving the equation $f(x) = 0$.

LIBRARY
Interpolation &
Differentiation
2

INTERPOLATION IN A FUNCTION KNOWN AT EQUAL INTERVALS OF ITS ARGUMENT

PROC equipol = (REF [] REAL fn, REAL t0) REAL

The values of the function are supplied as an array variable having arbitrary lower and upper bounds. The procedure uses all the given function values from the lower bound to the upper bound inclusive, and fits a polynomial of order n, where n is defined in the table below. It delivers the interpolated function value at a given intermediate value t0 of "t", which, for the purpose of the procedure, is regarded as the argument of the function and is defined as shown below:

t	function
0	fn[lower bound]
1	fn[lower bound + 1]
2	fn[lower bound + 2]
•	•••
n	fn[upper bound]

As an example, suppose that a is an array variable with bounds 1:100 (say), and suppose that a[3] = 1, a[4] = 4, a[5] = 9. Then the call

equipol(a[3:5], 1.5)

delivers, by quadratic interpolation, the result 6.25.

INTERPOLATION AND DIFFERENTIATION IN A FUNCTION KNOWN AT EQUAL INTERVALS OF ITS ARGUMENT

PROC equidif = (REF [] REAL fn, REAL t0) REALPAIR

The procedure delivers a REALPAIR consisting of the interpolated function value as defined in equipol, together with the interpolated derivative of the function with respect to t, both at the point t0.

INTERPOLATION IN A FUNCTION KNOWN AT UNEQUAL INTERVALS OF ITS ARGUMENT

```
PROC interpol = (REF[ ]REAL arg, fn, REAL arg0) REAL
```

The array variables arg and fn refer to arrays of the argument and function respectively. These may have arbitrary lower and upper bounds, but must be of the same size. The procedure fits a polynomial to all the given values from the lower bound to the upper bound inclusive, and delivers the interpolated value at the argument arg0. Fault display AA INTERPOL MISMATCH when the arrays arg and fn are not the same size.

INTERPOLATION AND DIFFERENTIATION IN A FUNCTION KNOWN AT UNEQUAL INTERVALS OF ITS ARGUMENT

```
PROC interdif = (REF[ ]REAL arg, fn, REAL arg0) REALPAIR
```

The procedure delivers a REALPAIR consisting of the interpolated function value as defined in "interpol", together with the interpolated derivative of the function with respect to the argument, both at the same point arg0. Fault display AB INTERDIF MISMATCH when the arrays arg and fn are not the same size.

LIBRARY

Interpolation &

Differentiation

4

SOLUTION OF $F(X) = 0$

```
PROC inverlate = (PROC(REAL)REAL fn, REAL xa, xb,
                   PROC REAL p) REAL
```

If $fn(xa)$ and $fn(xb)$ are of opposite sign, the procedure delivers a real argument x_0 which makes $fn(x_0) = 0$. If $fn(xa)$ and $fn(xb)$ have the same sign, inverlate calls the procedure p and delivers its result as its own result. The actual parameter for p can, however, be of the form GOTO label, which will then be coerced to mode PROC REAL to satisfy the formal parameter specification. This permits escape to a given program label when the procedure has been mis-applied.

REAL ROOTS OF A POLYNOMIAL EQUATION

```
PROC real roots = (REF[ ]REAL a)INT
```

This procedure delivers the number r of real roots of the equation

$$c_0 + c_1x + c_2x^2 + \dots + c_nx^n = 0$$

where the coefficients c_0 to c_n are given by the user in the parameter a. The array of coefficients may have any bounds. If $r > 0$, the values of the roots in numerical order (negative to positive) are assigned to the array elements

$a[LWBa + 1]$ to $a[LWBa + r]$ inclusive

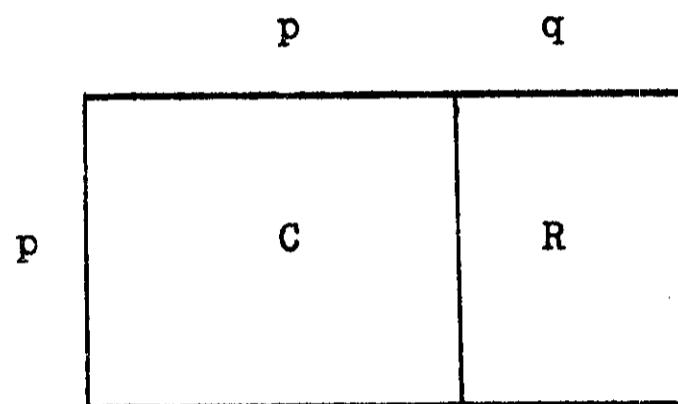
Multiple roots are each included the appropriate number of times.

MATRICES

SOLUTION OF SIMULTANEOUS EQUATIONS

```
PROC simeq = (REF[ , ]REAL a)REAL
PROC simeqcomp = (REF[ , ]COMPLEX a)COMPLEX
```

These procedures solve q sets of p simultaneous equations with real or complex coefficients respectively. The p by (p+q) matrix, a, may be regarded as made up of a p by p matrix C on the left, consisting of the coefficients, which are the same for every set of equations, with a p by q matrix R on the right made up of the q right-hand sides as additional columns.



The procedure alters the whole of the matrix in the process, and delivers the determinant of *C* as its value. The solution overwrites *R*, each column of which becomes the solution corresponding to the right-hand side originally occupying this position. If *q* = 0, no equations are solved, but the procedure delivers the correct value of the determinant, even if this is zero. Singular equations cause the fault display AE SINGULAR MATRIX.

SOLUTION OF FURTHER SIMULTANEOUS EQUATIONS WITH SAME COEFFICIENTS

```
PROC simeq repeat = (REF[ , ]REAL a)REAL
PROC simeqcomp repeat = (REF[ , ]COMPLEX a)COMPLEX
```

After simeq or simeqcomp has been called to solve a set of simultaneous equations, a new set of right-hand sides *R* may be supplied, and provided that *C* is not altered, simeq repeat (simeqcomp repeat) will replace *R* by the new solution vectors corresponding to the new right-hand sides, using working data left in *C* by simeq (simeqcomp).

MATRIX MULTIPLICATION

```
PROC matmult = (REF[ , ]REAL a, b, c)
PROC matmultcomp = (REF[ , ]COMPLEX a, b, c)
```

These procedures are for the multiplication of real and complex matrices respectively. If the matrices to which a, b and c refer are compatible in size, the procedure assigns the product of a and b to c, otherwise displays the fault MATRICES NOT COMPATIBLE.

DIAGONALISATION OF A REAL SYMMETRIC MATRIX

```
PROC symmetric qr = (REF[ , ]REAL a, REF[ ] REAL lambda,  
INT limit) INT
```

Evaluates the eigenvalues and eigenvectors of the square symmetric matrix a. On exit the eigenvalues are stored in lambda in ascending order and the corresponding (orthonormal) eigenvectors in the columns of a.

Fault display ARRAY BOUNDS DO NOT MATCH occurs if a is not square or the length of lambda is not equal to the dimension of a.

The method used is Householder's tridiagonalisation followed by QR iteration. The parameter "limit" sets the maximum number of QR iterations that will be attempted at any stage; a suitable value, which should be adequate in virtually every case, is 30. The result delivered by the procedure is the maximum number of iterations actually used. If this result is equal to limit, a and lambda will contain rubbish.

Accuracy should be close to machine rounding error (usually better than 1 in 1&10).

The procedure is a version of two procedures, tred2 and tq12, published in Numerische Mathematik 11 pp 181-195 (1968) and 11 pp 293-306 (1968) respectively and reference to those papers can be made for details of the method and its limitations.

DIAGONALISATION OF A HERMITIAN MATRIX

```
PROC hermitian qr = (REF[ , ]COMPLEX a, REF[ ]REAL lambda,  
INT limit) INT
```

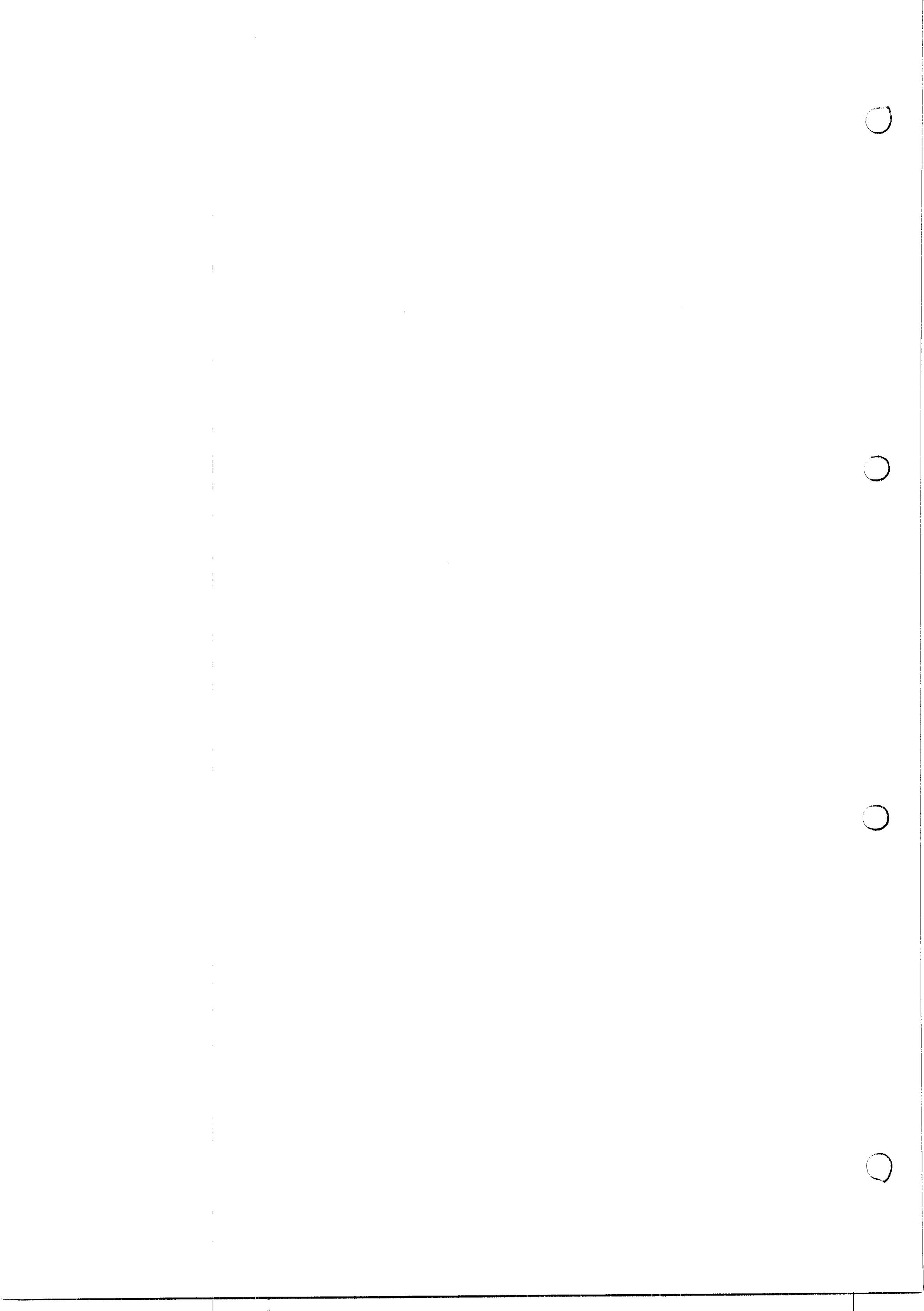
This procedure diagonalises the hermitian matrix a. On exit the eigenvalues of a are stored in ascending order in lambda and the corresponding orthonormal eigenvectors in the columns of a.

Fault display ARRAY BOUNDS DO NOT MATCH is produced if a is not square or the length of lambda is not equal to the dimension of a.

The diagonalisation is carried out by first reducing the starting matrix to tridiagonal form by Householder's method, and then using QR iteration (with explicit origin shifts) to evaluate the eigenvalues and back transformation to generate the eigenvectors.

The parameter "limit" sets the maximum number of QR iterations that will be carried out at any stage. A reasonable value for "limit" is 30; this is more than enough to give convergence in almost every case, but is small enough to save computing time in the remote chance that the matrix has some pathological properties.

The result delivered by "hermitian qr" is the maximum number of QR iterations actually used at any stage. If this result is equal to limit, a and lambda will contain rubbish.



SCALAR PRODUCT OF TWO VECTORS

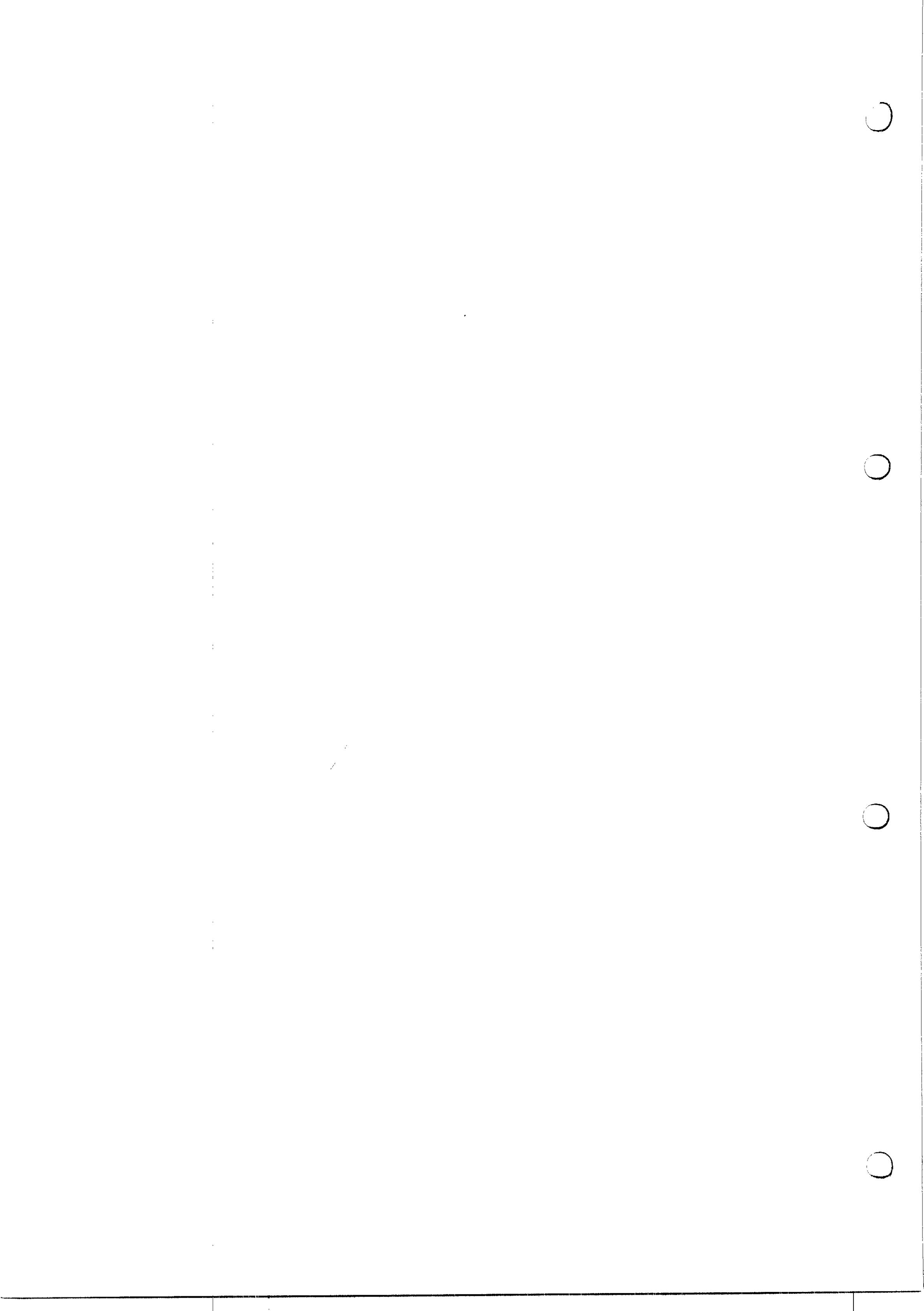
```
PROC scapro = ([]REAL a,b) REAL  
PROC scapro comp = ([]COMPLEX a,b) COMPLEX
```

These procedures deliver the scalar product of two real or two complex vectors respectively. The arrays a and b may have any bounds, provided that they are the same size.

HERMITIAN PRODUCT OF TWO COMPLEX VECTORS

```
PROC herpro = ([]COMPLEX a,b) COMPLEX
```

Delivers the Hermitian product of two complex vectors a and b, ie the scalar product $a^*.b$ where a^* is the vector whose elements are the complex conjugates of the elements of a. The arrays a and b may have any bounds, provided that they are the same size.



ADDING A FRACTION OF A ROW OF REALS TO ANOTHER ROW

```
PROC addfrac = (REAL x, [ ]REAL r, REF[ ]REAL rr) VOID
```

This procedure adds x times each element of the row r to the corresponding element of the row to which rr refers, where the rows may have any bounds provided they are the same size. The elements of r and those referred to by rr must not be overlapping parts of the same array, but rr may refer to r itself. In this case the procedure will scale each element of the row by the same factor, eg

```
addfrac(x, rr, rr)
```

would multiply each element of the row to which rr refers by $(x + 1)$. This is not recommended if the factor $(x + 1)$ is numerically less than 0.5 owing to possible loss of accuracy.

Fault display ARRAYS DO NOT MATCH IN PROC ADDFRAC occurs if the rows are not the same size.

Fault display OVERFLOW IN LIBRARY PROC ADDFRAC occurs if floating-point overflow is set during the procedure.

TRANSPOSING A REAL MATRIX

```
PROC transpose = (REF[,]REAL a) REF[,]REAL
```

This procedure transposes the real matrix a, and delivers a reference to the transposed matrix. The bounds for the two dimensions of "a" are interchanged by the procedure. Thus, after obeying the piece of program

```
INT p, q, r, s; read((p, q, r, s));  
[p:q, r:s]REAL a, adash;  
read(a); transpose(adash := a)
```

the bounds of adash will be altered to [r:s, p:q], and adash[i,j] will equal a[j,i] for all values of i and j within the bounds. Obeying transpose(transpose(a)) would leave a unaltered.

DATES

A date can be held in four different forms:

As an integer equal to the number of days elapsed since 31 DECEMBER 1899, for example, 26085 represents the date 2 JUNE 1971. This is a convenient form when dates are to be compared.

As an INTRIPLE with three integer fields (i,j,k) equal to the day of the month, month of the year and last two digits of the year respectively, for example (2,6,71).

In character form as a LONG BYTES, where the day, month and year are each represented by two digits, and are separated by slashes, for example "02/06/71".

As a LONG BYTES in the format used by George 3, namely, two digits or a space and a digit for the day, the first three letters of the month, the last two digits of the year and a final space, for example " 2JUN71 ".

Each of the following library variables is initialised to the appropriate form of the date when a program is first started.

```
INT days;  
INTRIPLE date;  
LONG BYTES datechars;  
LONG BYTES geo3date;
```

The four variables can be altered simultaneously to the alternative forms of some other date by using one of the following procedures:

```
PROC set days = (INT number)
```

This sets "days" equal to number, and sets the other variables to represent the same date. The number must be greater than 0 and must not exceed 36524 (corresponding to 31 DECEMBER 1999).

```
PROC set date = (INTRIPLE daymonthyear)
```

This sets "date" equal to the INTRIPLE parameter, and the other variables to correspond. The year of the date (the third field) may be in the range 0 to 99 or 1900 to 1999 inclusive; in the latter case only the last two digits will be stored in "k OF date".

```
PROC set datechars = (LONG BYTES bb)
```

This sets "datechars" equal to bb, and the other variables correspondingly. The eight characters of bb must consist of three groups each of two digits (not spaces) with any separating character between each group.

```
PROC set geo3date = (LONG BYTES bb)
```

This sets "geo3date" equal to bb, and the other variables to correspond. The eight characters of bb must be in the George 3 format described above, except that any character may be substituted for the final space.

Any one of the variables can be altered separately by simple assignment.

DAY AND MONTH IN CHARACTER FORM

The day of the week and the month in character form are obtained from the following procedures:

```
PROC day = INT  
PROC daychars = BYTES
```

These procedures use the value of the integer variable "days" (see Miscellaneous 1) which may be positive or negative and is taken as the date in days relative to 31 DECEMBER 1899. The procedure "day" delivers an integer in the range 1 (Sunday) to 7 (Saturday) and "daychars" delivers a bytes value consisting of a space followed by the first three characters of the day of the week.

```
PROC monthchars = BYTES
```

This delivers the month of the year corresponding to the integer "j OF date" (see Miscellaneous 1) as a bytes value - one space followed by the first three characters of the month. Fault display ILLEGALLY INITIALISED PROC MONTHCHARS means that j OF date does not lie in the range 1 to 12.

TIME OF DAY

PROC seconds now = INT

PROC timechars now = LONG BYTES

These procedures deliver the time of day to the nearest second, in different forms. The procedure "seconds now" gives the time in seconds measured from the previous midnight. The eight characters delivered by "timechars now" give the current time (using the 24-hour clock) in hours, minutes and seconds, each represented as two digits and separated by full stops, for example "16.12.42".

For converting times less than 60 hours from one form to the other

PROC seconds = (LONG BYTES time in chars)INT

delivers the time in seconds corresponding to the character form "time in chars". The separators in "time in chars" need not be full stops, but can be any characters.

PROC timechars = (INT secs)LONG BYTES

delivers the time in character form corresponding to the integer "secs" which must lie in the range 0 to 215999.

TERMINATING A PROGRAM AT ANY POINT

PROC free = VOID

This procedure jumps to the end of the program. It is particularly useful for terminating a complete program from inside a segment.

OBEYING A GEORGE COMMAND FROM A PROGRAM

```
PROC obey command = ([ ]CHAR s)BOOL:
```

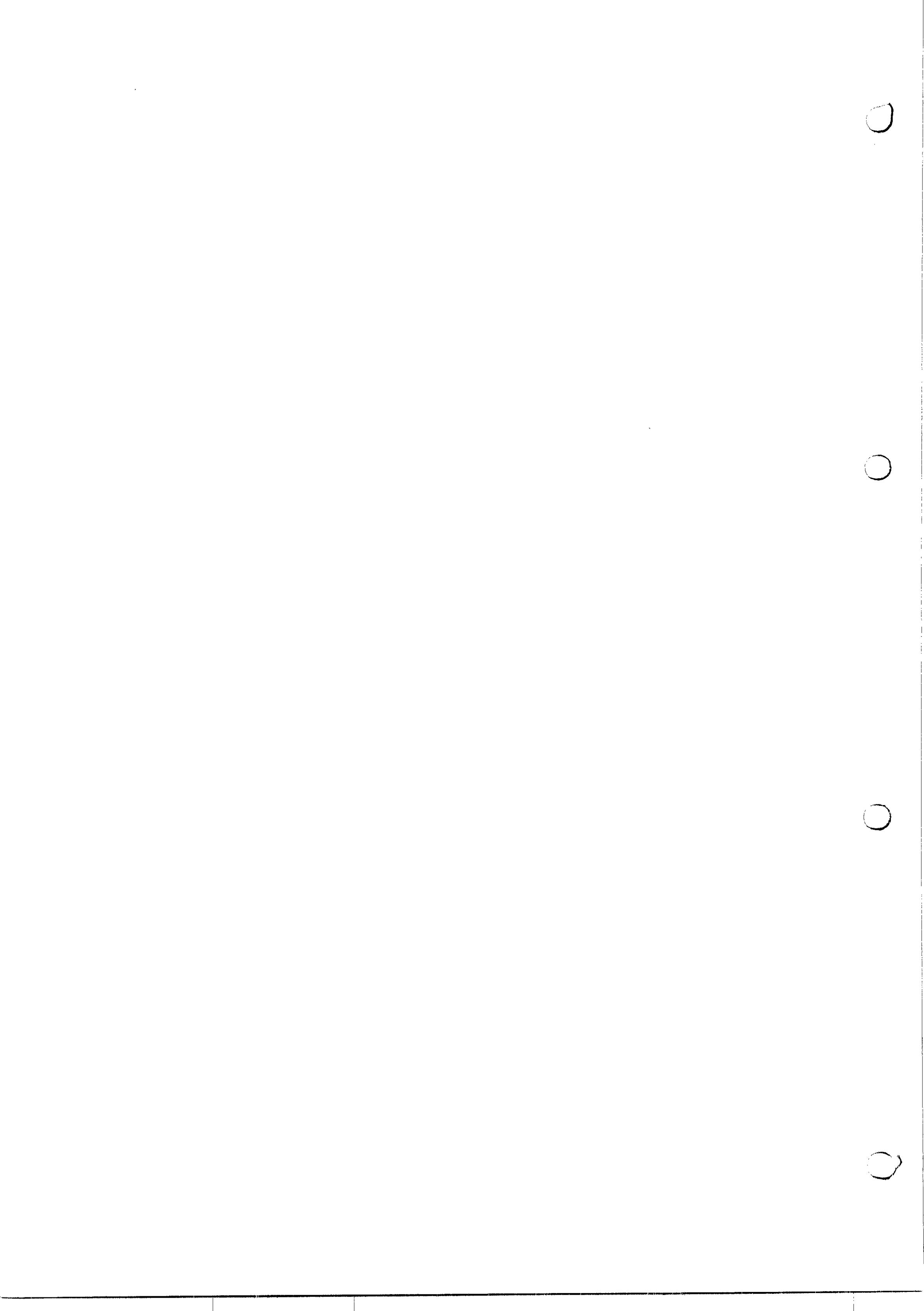
This procedure should be avoided if possible, as it may take several minutes to obey a George command from a program. The row-of-character parameter s is the command to be obeyed; the actual array supplied may have any bounds but must not contain more than 511 characters. Not all George commands can be given by a program. The principal use for the procedure is to obey ASSIGN commands if it is essential to associate several different files with one input/output channel during the running of the program, for example

```
obey command ("ASSIGN *fr1,secondfile")
```

After the command has been obeyed, the program continues normally. If the command is in error, the usual George error message is sent to the monitoring file of the job. The action of the program after this depends on the value of the library variable

```
BOOL comerrfault;
```

which is automatically initialised to TRUE and will cause fault display ILLEGAL PARAMETER OF PROC OBEY COMMAND. If "comerrfault" is set to FALSE, "obey command" will deliver TRUE if the command is obeyed successfully or FALSE if a command error occurs, and the program will continue normally.



TESTING SWITCHES

Every program has a set of 'switches', ie a group of distinct boolean values numbered from 0 to 23, any of which can be set by a George command (see GEORGE 4.2.7). A switch can be tested in the program by the procedure

```
PROC switch = (INT n)BOOL
```

which delivers TRUE if switch n is 'on', or FALSE if it is not. There are also two procedures for setting switch n on or off in the program, namely

```
PROC on = (INT n)
```

```
PROC off = (INT n)
```

The complete set of switches is held in the library variable

```
BITS switches;
```

which can be used if simultaneous testing or setting of a number of switches is required. Remember that the elements of a BITS value are numbered from 1 to 24.

Switch 23 should not be used if the program includes the library procedure "save".

OUTPUTTING A MESSAGE TO THE MONITORING FILE

PROC display = ([]CHAR message)

This procedure prints its message parameter on the monitoring file of the job, in the form

DISPLAY: message

The message must not contain more than 40 characters.

FORCING A PROGRAM FAILURE

PROC fault = ([]CHAR message)

This procedure prints its message parameter on the monitoring file in the same way as "display", and follows it with HALTED ER. It then performs the normal program failure routine, ie the current line of the standard input and output channels is printed, followed by a report of the position of the fault and current values of integer, real and complex variables. The whole job is then ended. The procedure is the same as that used by various library procedures when an error is detected.

PROC wrong = ([]CHAR message)

This procedure prints its message parameter on the monitoring file in the same way as display, and then jumps to the end of the program, with DELETED ER being sent to the monitoring file. No diagnostics are produced and the job is not ended; it will continue with the next George command after EXECUTE. Care should be taken that this does not lead to a series of consequential errors in the rest of the job.

EXTRA ACTION AT PROGRAM FAILURES

The standard action taken at program failures produces diagnostics consisting of the current line of the standard input and output channels and a report giving textual information on the position of the failure, together with current values of integer, real and complex variables and parameters. It is possible to obey a piece of program before printing these diagnostics by using

```
PROC postmortem = (PROC VOID p)VOID
```

If postmortem has been called with parameter p, at the start of a program for example, the action taken at a subsequent program failure will be to obey the procedure p. Typically this would be used to print values in arrays or structures which are not printed by the standard diagnostic routine. When p has been obeyed, HALTED PM occurs on the monitoring file, after which normal diagnostics for the failure will be printed. If there is another failure during the running of p, normal diagnostics are produced for the second failure, showing the postmortem procedure to have been called from the position of the first failure, so that both failure positions can be found. If there is a jump out of p, the original failure will be ignored, except that postmortem cannot be called again, and any further failure will cause normal diagnostics to be printed.

Due to the nature of some failures, postmortem cannot always be expected to work successfully. Its effect can be cancelled by calling

```
PROC reset report = VOID
```

which restores the standard failure action. This must always be done if the procedure given in the call of postmortem goes out of scope.

CONTINUING AFTER A PROGRAM FAILURE

A program which has already been debugged may fail under particular known circumstances for which it is impractical to check. The program can be made to ignore the fault and continue with a different part of the computation by using the procedure

```
PROC revive = (PROC VOID g)VOID
```

This procedure should be called as near as possible to the place where the program is expected to fault. Its parameter should be a GOTO, to another part of the program. The procedure "reset report" (Miscellaneous 8) must be called after the possible failure point, to restore the standard diagnostic action should the expected fault not occur.

The revive procedure only affects program failures which end in the standard library fault procedure of displaying an error message on the monitoring file followed by HALTED ER. If revive has been called, HALTED ER will not occur and the program will continue by obeying the jump g after resetting the standard failure action. Revive may be called again, but the number of faults trapped in this way should be limited.

Other failures, such as trying to obey illegal instructions, are treated in the standard way, diagnostics being produced. However postmortem may be used in addition to revive, in which case these other failures will invoke the postmortem procedure. Both postmortem and revive are automatically cancelled after a failure has been dealt with, and if a postmortem has occurred neither can be called again.

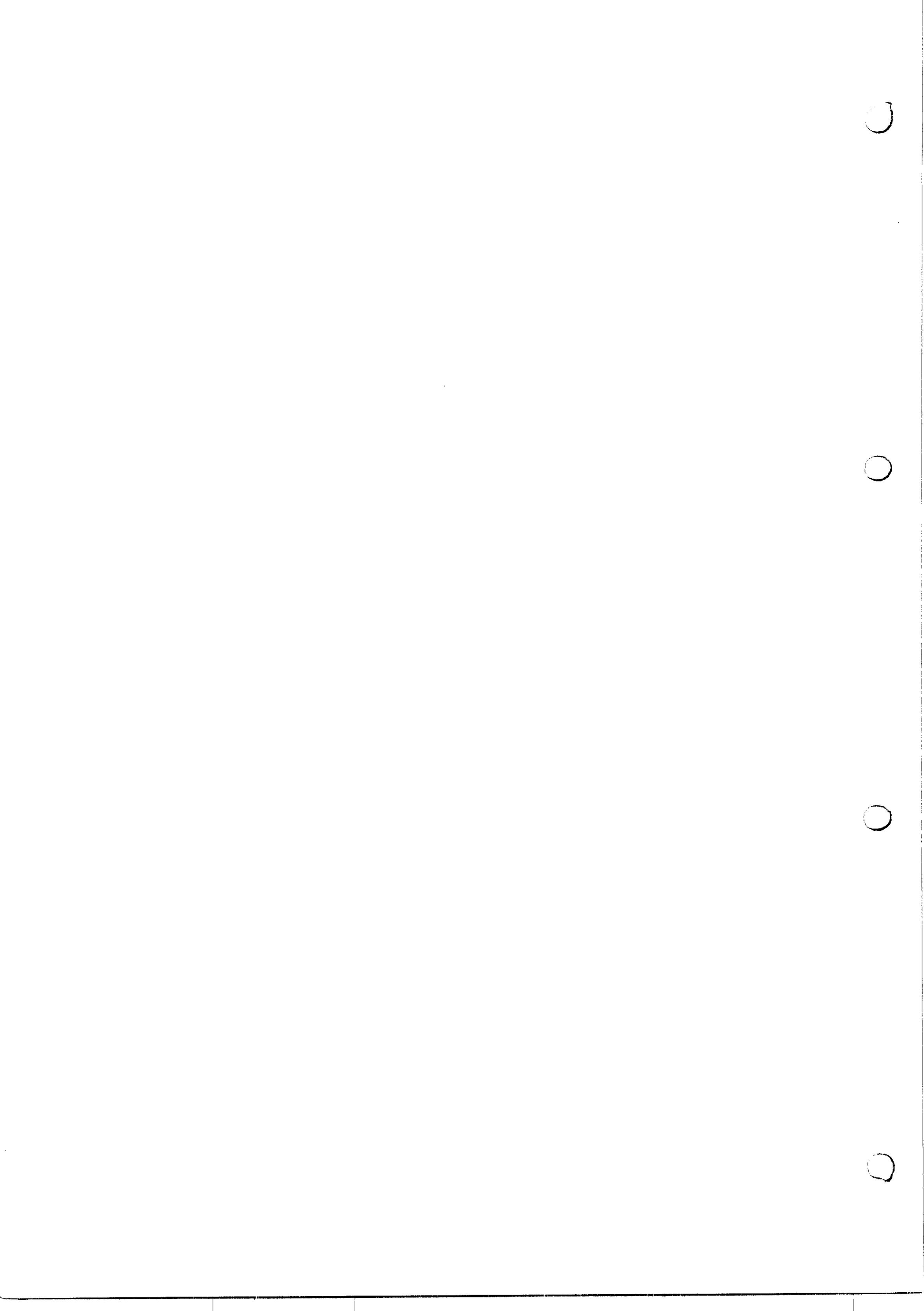
MODES

```
MODE COMPLEX = STRUCT (REAL re, REAL im)
MODE INTPAIR = STRUCT (INT i, INT j)
MODE INTREAL = STRUCT (INT i, REAL x)
MODE INTRIPLE = STRUCT (INT i, INT j, INT k)
MODE REALPAIR = STRUCT (REAL x, REAL y)
```

The above simple structures of integers and reals are used in some library procedures (eg autoquad) and may be assumed to have been declared in every program.

```
MODE STRING = [1:0 FLEX] CHAR
```

Use of STRING should be carefully considered, as it will involve off-stack storage allocation and consequent time and space overheads. Simple arrays of characters will meet the requirements of most applications.



OPERATORS

The following is a list of operators, grouped according to the type of object operated on. Within each group they are arranged in order of decreasing tightness of binding, the first in each group having the highest priority and binding the tightest.

A standard operator symbol may be

a single character

eg ? Both <= and >= are treated as single characters when used as operators.

an upper case word not already used for some other purpose

eg ENTIER These words are enclosed in primes if graphic conventions are being used.

several symbols enclosed in primes to group them

eg '/+' The primes are necessary in both normal and graphic conventions.

The integer following the operator gives its exact priority. Priorities of dyadic operators are in the range 1 to 9; the highest priority (10) applies only to monadic definitions.

LIBRARY
Operators
2

ARITHMETICAL

- ↑ 8 x^n gives x raised to the power n , where n must be an integer and x may be real (giving a real result) or integer (giving an integer result which is 0 if $n < 0$). To raise a complex number to the power n , see LIBRARY Functions 10.
- * / 7 times, divide between integers, reals and complex numbers in any combination, and between long integers. Integer divided by integer always gives a real result.
- '/' 7 used between integers n, m gives an integer result which is ENTIER (n/m). Similarly for long integers, giving a long integer result.
- + - 6 plus, minus between integers, reals and complex numbers in any combination, and between long integers. Can also be used monadically (priority 10).

ARITHMETICAL COMPARISONS

These operators give a BOOL result depending on whether the relation expressed is true or false.

- < <= 5 less than, less than or equal,
> >= 5 greater than, greater than or equal
 between integers and reals in any combination and
 between long integers
- = # 4 equals, not equals between integers and between
 long integers but not between reals

ARITHMETICAL ASSIGNMENTS

Each operator in this group performs an arithmetic operation and assigns its result to the left-hand operand.

PLUS 1 x PLUS y means $x := x + y$

x may be REF INT, REF REAL or REF COMPLEX, and $x + y$ must deliver a result suitable for assignment. (For example, if x is REF INT, y must not be REAL.)

MINUS 1 x MINUS y means $x := x - y$

Similar to PLUS.

TIMES 1 x TIMES y means $x := x * y$

x must be either REF REAL or REF INT and $x * y$ must deliver a result suitable for assignment. (For example, if x is REF INT, y must not be REAL.)

DIV 1 x DIV y means $x := x/y$

x must be REF REAL and y may be real or integer. An alternative for DIV is OVER.

'/:=' 1 used between n and m where n is REF INT and m is an integer.
or

'/-' 1 n '/-' m performs the assignment $n := \text{ENTIER}(n/m)$ and delivers the remainder with the same sign as m. For example

```
INT n := -7, rem;  
      rem := n'/-' 3
```

would assign -3 to n and +2 to rem.

OTHER NUMERICAL OPERATORS

ABS 10 absolute value, or modulus of, an integer, a real or a complex. For complex operands, overflow will occur if the square of the modulus overflows.

SIGN 10 applied to an integer or a real, x, gives the integer value

+1 if x is greater than zero
0 if x is zero
-1 if x is less than zero

ODD 10 applied to an integer, gives the boolean value TRUE if the integer is odd, and FALSE if it is even.

ENTIER 10 applied to a real, gives as its result the largest integer less than or equal to the real.

ROUND 10 applied to a real, gives as its result the nearest integer to the real. In the critical case, it rounds up.

LENG 10 applied to an INT value, delivers the LONG INT value.

SHORT 10 applied to a LONG INT value, delivers the INT value.

? 9 used between x and y, each of which may be integer or real, delivers the complex number whose real part is x and whose imaginary part is y.

ARG 10 applied to COMPLEX delivers the argument, ie phase, in the range (-pi,pi), mode REAL.

CONJ 10 applied to COMPLEX, delivers the conjugate.

OPERATIONS ON BOOLEANS

- | | | |
|-----|----|---|
| NOT | 10 | delivers TRUE if the operand is FALSE and vice versa. |
| AND | 3 | delivers TRUE if both operands are TRUE, otherwise delivers FALSE. The right operand will only be evaluated if the left operand has the value TRUE. |
| OR | 2 | delivers TRUE if at least one operand is TRUE, otherwise delivers FALSE. If the left operand has the value TRUE, the right operand will not be evaluated. |
| = # | 4 | equals, not equals between booleans, delivers TRUE if the relation is TRUE, otherwise delivers FALSE. |
| ABS | 10 | delivers the integer 1 if the operand is TRUE, and 0 if it is FALSE. |

ifc(9.75)

OPERATIONS ON ARRAYS

- LWB 10 applied to an array gives the lower bound of its first dimension (an integer result)
- UPB 10 applied to an array gives the upper bound of its first dimension
- m LWB p 8 where m is an integer and p an array gives the lower bound of the m-th dimension of p. Thus 1 LWB p gives the same as LWB p
- m UPB p 8 where m is an integer and p an array gives the upper bound of the m-th dimension of p
- FLEXIBLE 10 applied to any array, delivers TRUE if the array has flexible bounds, otherwise delivers FALSE
- CLEAR 10 applied to a reference to an array with any number of dimensions, "clears" the array. For the following modes of array element the resulting values are

INT	0
LONG INT	LONG 0
REAL	0.0
COMPLEX	0.0 ? 0.0
BOOL	FALSE
BITS	BIN 0
CHAR	space character
BYTES	"0000"
LONG BYTES	"00000000"
LONG LONG BYTES	"000000000000"

The elements may also be structures with fields having these modes. In this case the fields will take values by the same rules except that CHAR fields will be set to "0".

BIT PATTERNS

A programmer may often want to consider a whole computer word as a set of 24 binary digits. Such a set is declared to be of mode BITS, thus

```
BITS mask = 2r111000;  
BITS pattern := 8r40004000;
```

A BITS value occupies one word, and operations such as masking and shifting may be carried out on it. Each bit in the word has mode BOOL (the binary digit 1 being TRUE and 0 being FALSE) and may be accessed only by using the operator ELEM described below.

Where bit patterns are to be written explicitly in a program in an assignment or identity relation, integers may be used as shown above. These integers are automatically considered as the bit pattern they represent. Use of the 2, 4, or 8 radix notation may be convenient, particularly when the sign bit is involved. If an explicit bit pattern is required in an expression, the operator BIN may be used to change an integer value to a value of mode BITS.

CORRESPONDENCE BETWEEN BITS AND INTEGERS

- ABS 10 applied to a BITS value gives the (signed) integer it represents.
- BIN 10 applied to an integer gives as its result the pattern of mode BITS which represents that integer.

LOGICAL OPERATIONS ON BITS

Each of these operators gives a BITS result.

- NOT 10 applied to a BITS value, delivers the BITS value with every digit reversed.
- AND 3 when used between two BITS values p, q, each binary digit of the result is the logical "and" of the corresponding digits of p and q. May also be used between a BITS value and an integer.
- OR 2 when used between two BITS values p, q, each binary digit of the result is the logical "or" of the corresponding digits of p and q, ie 0 if both digits are 0, otherwise 1. May also be used between a BITS value and an integer.

LOGICAL AND CIRCULAR SHIFTING

- SL 7 b SL m where b is a BITS value and m is a positive integer gives as its result the 24-bit pattern b shifted left m places. Bits are lost on the left, and zeros created on the right. Overflow cannot be set.
- SR 7 similar to above, but shifting right.
- SLC 7 shift left cyclically. Operands as for SL, but binary digits lost on the left are introduced on the right.
- SRC 7 shift right cyclically, similar to SLC.

COMPARING BITS VALUES

- <= >= 5 less than or equal, greater than or equal between BITS values p, q delivers a BOOL result. p <= q is TRUE if each binary digit which is 1 in p is also 1 in q.
- = # 4 equals, not equals between BITS values, delivers TRUE if the relation is true and FALSE otherwise.

OPERATIONS ON PARTICULAR BINARY DIGITS

These operators regard the binary digits of a BITS value as being numbered from 1 to 24 inclusive, starting at the most significant end. The sign bit is thus bit 1.

- ELEM 7 m ELEM b where m is an INT and b is a BITS value, delivers TRUE if the m-th binary digit of b is 1, and FALSE if it is 0. An alternative for ELEM is TH.
- SET 7 m SET b where m is an INT and b a BITS value delivers the bit pattern b, but with the m-th binary digit as 1. Note that the operator does not change b.
- CLEAR 7 m CLEAR b where m is an INT and b a BITS value delivers the bit pattern b, but with the m-th binary digit as 0. Note that the operator does not change b.

LIBRARY
Operators
10

CHARACTERS

ABS 10 applied to a character gives its internal integer representation (in the range 0 to 63).

REPR 10 applied to an integer in the range 0 to 63 inclusive gives the character it represents

CHARACTER SET

0	0	1	2	3	4	5	6	7	8	9
10	:	;	<	=	>	?	sp	!	"	#
20	f	%	&	'	()	*	+	,	-
30	.	/	@	A	B	C	D	E	F	G
40	H	I	J	K	L	M	N	O	P	Q
50	R	S	T	U	V	W	X	Y	Z	[
60	\$]	↑	←						

COMPARING CHARACTERS AND SETS OF CHARACTERS

Sets of characters may be bytes, long bytes, long long bytes or one-dimensional arrays of characters (including strings). All the following operators deliver a BOOL result depending on whether the relation expressed is true or false.

< <= 5 less than, less than or equal
> >= greater than, greater than or equal
between characters and sets of characters as listed above in any combination. If c and d are characters

c < d if ABS c < ABS d

When comparing sets of characters, successive characters from each set are compared until a decision is made or the smaller set is exhausted. The ordering is lexicographic, ie

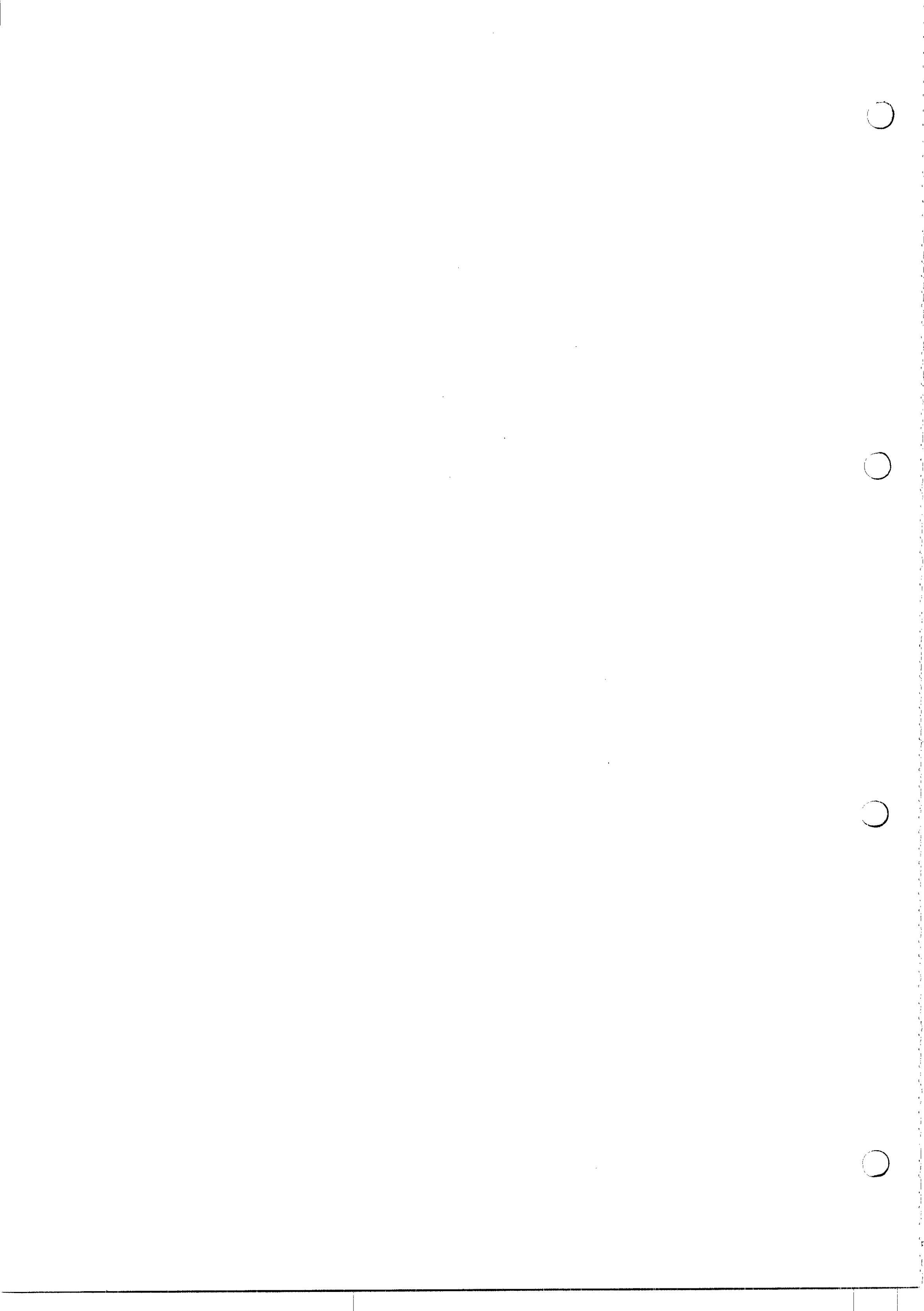
"AA" < "AAB" is TRUE,
and "AAB" < "AB" is TRUE

= # 4 equals, not equals between characters and sets of characters in any combination

SPECIAL BYTES OPERATORS

CTB 10 (characters to bytes)
CTLB (characters to long bytes)
CTLLB (characters to long long bytes)
applied to words of 4, 8 or 12 characters
respectively, gives as result the characters
as a bytes, long bytes or long long bytes value

ELEM 7 m ELEM c where m is an integer and c may be bytes, long
bytes or long long bytes gives as its result the m-th
character of c. The characters are numbered from
1 to 4, 8 or 12 respectively, starting from the left.
An alternative for ELEM is TH

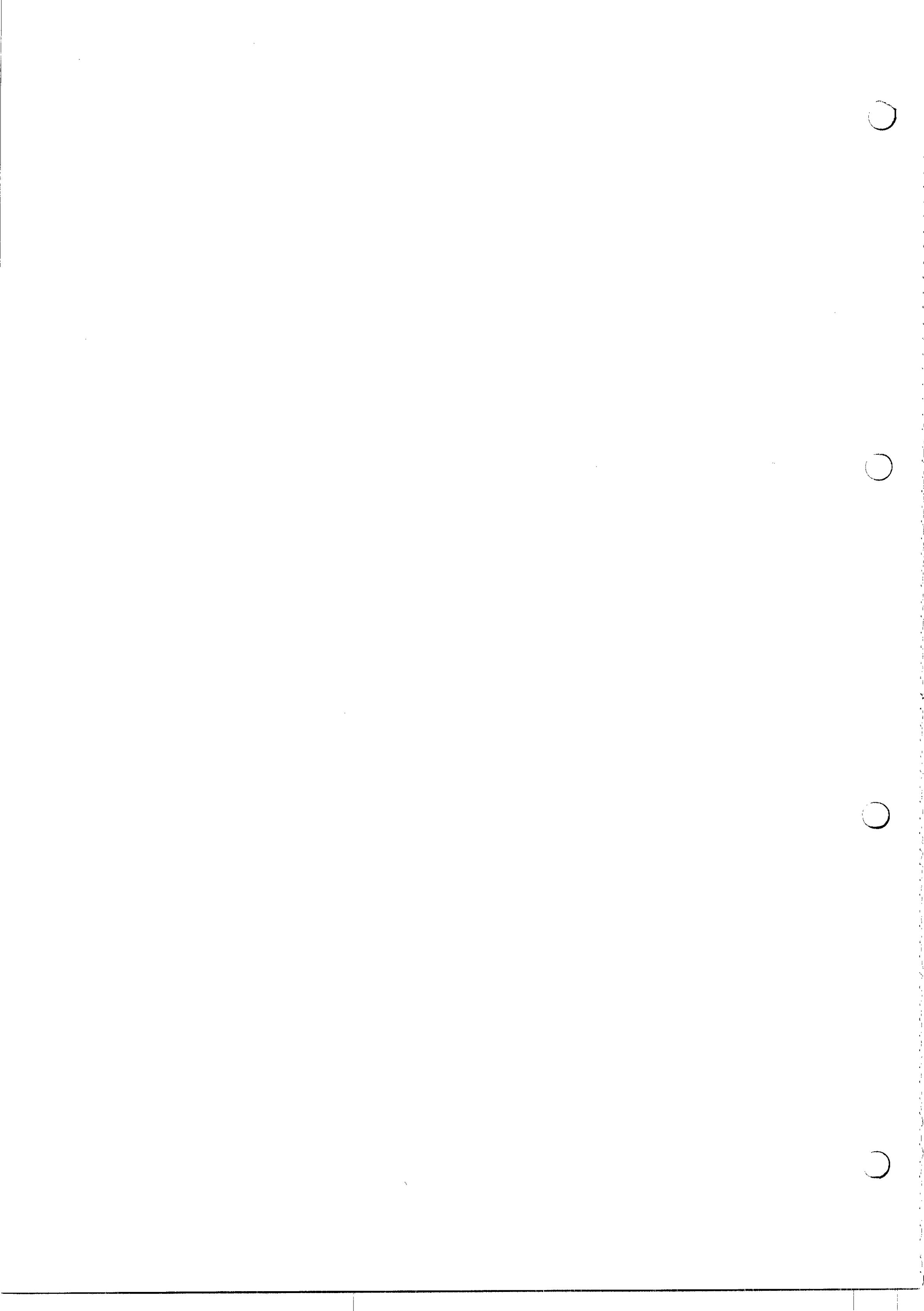


PRIORITIES OF OPERATOR SYMBOLS

Operators with high priority numbers bind more tightly than those with low numbers. Monadic operators always bind most tightly of all, and are shown with priority 10.

If one of the standard operator symbols is re-declared dyadically in a program and no priority declaration is given, the standard dyadic priority of the operator applies. If there is no standard dyadic priority, the default priority is 1, with the exception of ENTIER, which will have priority 8. For example, if NOT is declared as a dyadic operator, its default priority is 1 (not 10).

<u>Symbol</u>	<u>Priority</u>	<u>Symbol</u>	<u>Priority</u>
ABS	10	?	9
AND	3	↑	8
ARG	10	*	7
BIN	10	/	7
CLEAR	7 (dyadic) 10 (monadic)	'/'	7
CONJ	10	+	6 (dyadic) 10 (monadic)
CTB	10	-	6 (dyadic)
CTLB	10		10 (monadic)
CTLLB	10	<	5
DIV	1	<=	5
ELEM	7	>	5
ENTIER	10	>=	5
FLEXIBLE	10	=	4
LENG	10	#	4
LWB	8 (dyadic) 10 (monadic)	'/->' '/:=>'	1
MINUS	1		
NOT	10		
ODD	10		
OR	2		
OVER	1		
PLUS	1		
REPR	10		
ROUND	10		
SET	7		
SHORT	10		
SIGN	10		
SL	7		
SLC	7		
SR	7		
SRC	7		
TH	7		
TIMES	1		
UPB	8 (dyadic) 10 (monadic)		



QUADRATURE

Quadrature is the numerical analyst's term for evaluation of a definite integral. The integrand is computed at a number of discrete points between the lower and upper limits, a polynomial is fitted at these points, and the integral is obtained from the formula for the integral of the polynomial. All of this is performed automatically by a quadrature procedure.

In one type of procedure, the values of the integrand at specific points are supplied as an array of numbers - as for an interpolation procedure. In another type, of which "autoquad" and "multiquad" are examples, the procedure selects for itself the values of the argument at which it requires to have the integrand evaluated. Such procedures must be supplied with an algorithm for computing the integrand at any argument. This is done by first expressing the integrand itself as a procedure which will deliver the evaluated integrand as result, and supplying this procedure as a parameter to the library quadrature routine. The mode of the integrand procedure depends on which library routine is being used. For example, "autoquad" deals with real functions of one real variable, ie integrands of mode PROC(REAL)REAL. The procedure sin has mode PROC(REAL)REAL, so to integrate sin(x) with respect to x from 0 to pi, the call

```
autoquad(sin, 0.0, pi, standaccuracy)
```

is correct, and delivers the result 2.0. Computing time can be saved by giving a less stringent accuracy criterion as the fourth parameter (see Approximation 1).

DEFINITE INTEGRAL OF A FUNCTION

```
PROC autoquad = (PROC(REAL)REAL f, REAL lower, upper,  
                  REALPAIR accuracy) REAL
```

Delivers an approximation to the definite integral of $f(x)$ with respect to x , from the lower limit "lower" to the upper limit "upper", with an accuracy controlled by the REALPAIR parameter. The x-field of this parameter is a figure for the tolerable fractional error, and the y-field a figure for tolerable absolute error. The procedure attempts to deliver its result so as to satisfy the least stringent of these two criteria (see Approximation, 1). The method used for integration is a 4-point Gauss Rule, in which the step sizes are adapted to the shape of the function f . These steps will never be smaller than about 1/100,000 of the total integration range. If the function f has discontinuities, much time will be wasted, and it is therefore best to split up the integration and make separate calls of autoquad for each smooth portion of the function.

MULTIPLE QUADRATURE OVER HYPER-RECTANGULAR REGION

PROC multiquad = (PROC([]REAL)REAL fn, []REAL lower, upper,
REALPAIR accuracy) REAL

Delivers an approximation to the value of the multiple definite integral

$$\int_{a_n}^{b_n} \cdots \int_{a_2}^{b_2} \int_{a_1}^{b_1} f(v_1, v_2, \dots v_n) dv_1 dv_2 \cdots dv_n$$

where f is a real function of real variables given by the parameter fn . This must be a procedure taking a []REAL parameter, say x with bounds 1 to n , and delivering $f(x[1], x[2] \dots x[n])$ as result. There is no advantage in ordering the variables of integration in the array in any particular way. The corresponding limits of integration, $a_1 a_2 \dots a_n$ and $b_1 b_2 \dots b_n$ (all being independent of the variables of integration) are supplied as the elements of the array parameters lower and upper. These may have any bounds provided that they are the same size; the size determines the number of dimensions of the integral. The relative and absolute accuracies demanded are given by the REALPAIR parameter accuracy (see Approximation 1). The procedure attempts to satisfy the least stringent of these criteria, but the accuracy may be limited by round-off errors if too high an accuracy is demanded.

The procedure should only be used to integrate functions which are continuous and have continuous derivatives. If the function or its derivatives have any internal discontinuities or infinities, the region should be split up into two or more regions over each of which the function and its derivatives are continuous. However, because the procedure only uses values of the function at interior points of the region, it may be used to evaluate integrals of functions which become indeterminate or even infinite at points on the boundary of the region, although convergence may be slower in the latter case.

In theory integrals of any number of dimensions may be evaluated, but in practice this may be limited by the time required. The minimum number of function evaluations required for an n -dimensional integral is $(2n + 1) \cdot 4^n$ which should give a relative accuracy better than one part in a million for reasonably well-behaved functions. The time for a fairly complicated 4-dimensional integral at this accuracy would be of the order of 10 seconds. This will normally be much faster than nested calls of autoquad.

MULTIPLE COMPLEX QUADRATURE OVER HYPER-RECTANGULAR REGION

PROC multiquad comp = (PROC([]REAL)COMPLEX fn, []REAL lower, upper,
REALPAIR accuracy) COMPLEX

This procedure is similar to multiquad, the integrand being a complex function of real variables.

RANDOM NUMBER GENERATION

The procedures for delivering pseudo-random numbers are all based on the use of "m-sequences". Each call of a procedure such as "random" produces a number which is, as far as practicable, statistically independent of what it has delivered before, but the whole sequence repeats after 8388607 calls of the same procedure (except for "gaussian", whose period is twice this number). When a program using "random" (say) is run more than once, different random numbers will be obtained each time unless the re-setting variable associated with the procedure is explicitly initialized by the programmer. The mode of this variable is different for different procedures; in simple cases it is a REF INT, in others it is a REF STRUCT containing an INT field. The re-setting integer changes whenever the procedure is called, and a repeatable starting point is obtainable by assigning an integer in the range 1 to 8388607 (inclusive) before the procedure is first called. Each one of these integers corresponds to a different starting point. If zero is assigned to the re-setting variable, the procedure selects its own starting point (using the time of day in seconds), but if no assignment is made, this happens automatically every time the user's program is executed.

If two or more sequences are required from a particular procedure in one program, for example to enable separate initializations to be performed for each sequence, more than one re-setting variable will be needed. As the library provides only one of these for each procedure, the user will have to declare his own (with the mode specified in the description of the procedure). The procedure will use whichever of these variables was last assigned to a reference variable associated with that procedure. Calls of the procedure with one re-setting variable in the reference variable are completely independent of calls with another in it. If no assignment is made to the reference variable, the library re-setting variable is assigned by default.

LIBRARY

Random

2

RANDOM NUMBERS UNIFORMLY DISTRIBUTED IN (0.0, 1.0)

PROC random = REAL

Successive calls of random deliver real numbers randomly and independently distributed in a rectangular distribution from 0.0 to 1.0. During the period, each of the numbers $r/8388608$ occurs exactly once, where r is an integer in the range 1 to 8388607 inclusive.

The library segment automatically declares and initializes the re-setting variable "normrand" thus

INT normrand := 0

To start the sequence in a repeatable manner, a positive integer should be assigned to normrand before the first call of random. This fixes the starting point in the random sequence. Zero gives an unrepeatable sequence.

A common application of random is to effect a jump with a certain probability. Thus, for example,

IF random < 0.3 THEN GOTO label FI

causes a jump to label with probability 0.3. See also Random 3.

To obtain a number of entirely distinct sequences in one program, it is necessary to use the library reference variable "randvar" associated with this procedure. Internally, random uses randvar as a reference to normrand, which it does not use directly. The library declaration of randvar is

REF INT randvar := normrand

For distinct sequences, other integer variables p, q, r (say), declared in the user's program, may be assigned to randvar in place of normrand. After an assignment such as

randvar := p

all calls of random use p as the re-setting variable. When such variables are used, they must always be initialized before the first calls of random associated with them. For repeatable starting, a positive integer must be assigned, otherwise zero.

RANDOM INTEGERS UNIFORMLY DISTRIBUTED

```
PROC intrandom = INT
```

Successive calls of intrandom deliver 23-bit integers randomly and independently distributed over the range 1 to 8388607 inclusive, unless the range is restricted as described below.

The library segment automatically declares and initializes the re-setting variable "normintrand" thus

```
INTPAIR normintrand := (23, 0)
```

To start the sequence in a repeatable manner, a positive integer should be assigned to the second field (j OF normintrand) before the first call of intrandom. This fixes the starting point in the random sequence. Zero gives an unrepeatable sequence.

To obtain integers in the range 0 to $2^b - 1$, where b lies between 1 and 23 inclusive, the first field of normintrand (i OF normintrand) should be set to b when the sequence is initialized. During the period, which is always 8388607, each integer in the given range occurs an equal number of times, except zero which occurs one time fewer. The smaller the value of b, the faster the procedure works.

To obtain a number of entirely distinct sequences in one program, it is necessary to use the library reference variable "inrandvar", which is automatically declared in the form

```
REF INTPAIR inrandvar := normintrand
```

INTPAIR variables declared in the program can be assigned to inrandvar in place of normintrand, in a similar way to that described for the procedure "random" (see Random 2).

To obtain a BOOL with a probability $p = a/(2^b)$ of being TRUE, the expression (intrandom < a) may be used, the first field of the INTPAIR having been initialized to b. This is considerably faster than using (random < p).

LIBRARY

Random

4

RANDOM NUMBERS NORMALLY DISTRIBUTED

PROC gaussian = REAL

Successive calls of gaussian deliver real numbers randomly and independently distributed in a gaussian distribution with zero mean and unit variance (mean square).

The library segment automatically declares and initializes the re-setting variable "normgauss" thus

INTREAL normgauss := (0, 0.0)

To start the sequence in a repeatable manner, a positive integer should be assigned to the first field (i OF normgauss) before the first call of gaussian. This fixes the starting point in the random sequence. Zero gives an unrepeatable sequence. If normgauss is initialized by means of a collateral, the second field should be 0.0. If i OF normgauss is initialized more than once in the same program, the second field (x OF normgauss) must be set to zero at the same time.

To obtain a number of entirely distinct sequences in one program, it is necessary to use the library reference variable "gaussvar", which is automatically declared in the form

REF INTREAL gaussvar := normgauss

INTREAL variables declared in the program can be assigned to gaussvar in place of normgauss, in a similar way to that described for the procedure "random" (see Random 2). Both fields of these variables must be initialized for each sequence, as described for normgauss above.

The expression ($m + s * gaussian$) will give a random number from a gaussian distribution with mean m and standard deviation s .

RANDOM NUMBERS IN NEGATIVE EXPONENTIAL DISTRIBUTION

PROC negexp = REAL

Successive calls of negexp deliver real numbers randomly and independently distributed in a negative exponential distribution with unit mean and variance. The probability that the number lies in the small interval $(x, x + dx)$ is $\exp(-x) dx$, $x \geq 0$.

The library segment automatically declares and initialises the re-setting variable "normnegexp" thus

INT normnegexp := 0

To start the sequence in a repeatable manner, a positive integer should be assigned to normnegexp before the first call of negexp. This fixes the starting point in the random sequence. Zero gives an unrepeatable sequence.

To obtain a number of entirely distinct sequences in one program, it is necessary to use the library reference variable "negexpvar", which is automatically declared in the form

REF INT negexpvar := normnegexp

Integer variables declared in the program can be assigned to negexpvar in place of normnegexp, in a similar way to that described for the procedure "random" (see Random 2).

The expression $(k * negexp)$ will give a random number from a negative exponential distribution with mean k and variance k^2 , whose frequency function is $\exp(-x/k)/k$, $x \geq 0$.

LIBRARY
Random
6

RANDOM NUMBERS IN RAYLEIGH DISTRIBUTION

PROC rayleigh = REAL

Successive calls of rayleigh deliver real numbers randomly and independently distributed in a Rayleigh distribution derived from a circular normal bivariate distribution having zero mean and unit variance in each dimension. The probability that the number lies in the small interval $(r, r + dr)$ is $r \cdot \exp(-r^2/2) dr$, $r \geq 0$.

The library segment automatically declares and initialises the re-setting variable "normrayleigh" thus

INT normrayleigh := 0

To start the sequence in a repeatable manner, a positive integer should be assigned to normrayleigh before the first call of rayleigh. This fixes the starting point in the random sequence. Zero gives an unrepeatable sequence.

To obtain a number of entirely distinct sequences in one program, it is necessary to use the library reference variable "rayleighvar", which is automatically declared in the form

REF INT rayleighvar := normrayleigh

Integer variables declared in the program can be assigned to rayleighvar in place of normrayleigh, in a similar way to that described for the procedure "random" (see Random 2).

The expression $(k * rayleigh)$ will give a random number from a Rayleigh distribution whose frequency function is

$$(r/k^2) \cdot \exp(-r^2/2k^2) \quad (r \geq 0)$$

RANDOM INTEGERS IN POISSON DISTRIBUTION

```
PROC poisson = INT
```

Successive calls of poisson deliver integers randomly and independently distributed in a poisson distribution with mean (and variance) m , where m is any real number in the range (0.0, 178.0). The probability that the procedure delivers the value r is

$$\exp(-m) * (m^r)/r! \quad (r = 0, 1, 2, \dots)$$

If $m > 178.0$ approx, $\exp(-m)$ becomes zero and the procedure fails.

The library segment automatically declares the re-setting variable

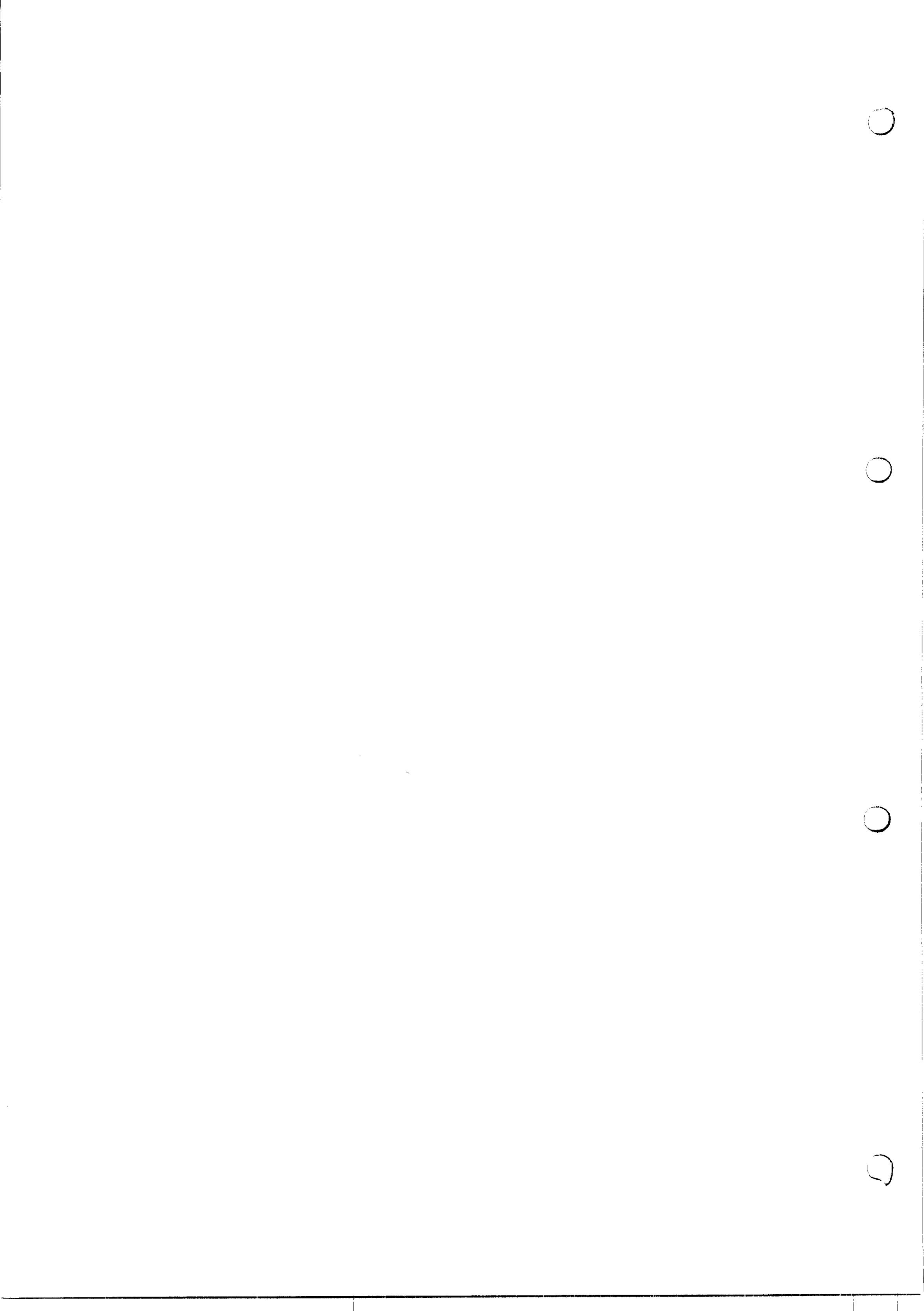
```
INTREAL normpoisson ;
```

The second field of this variable (x OF normpoisson) must be initialised to the mean m before the first call of poisson in any sequence. The first field (i OF normpoisson) is automatically initialised to zero, which gives an unrepeatable sequence. To start a sequence in a repeatable manner, i OF normpoisson must be initialised with a positive integer, which fixes the starting point in the random sequence.

To obtain a number of entirely distinct sequences simultaneously in one program, it is necessary to use the library reference variable "poissonvar", which is automatically declared in the form

```
REF INTREAL poissonvar := normpoisson
```

INTREAL variables declared in the program can be assigned to poissonvar in place of normpoisson, in a similar way to that described for the procedure "random" (see Random 2). Such variables must be suitably initialised as above before first use.

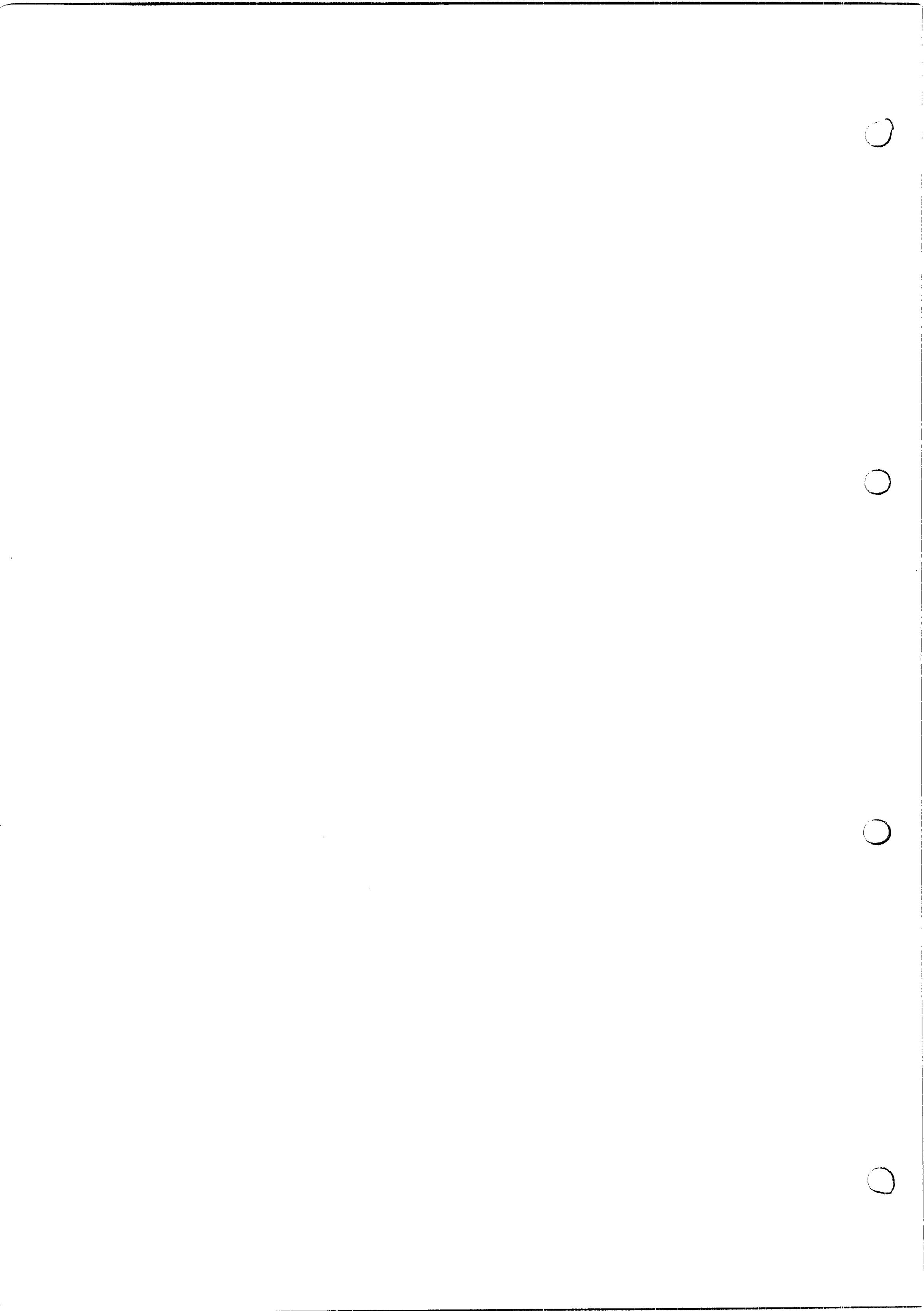


CHI-SQUARED PROBABILITY DISTRIBUTION FUNCTION

```
PROC chiprob = (REAL x, INT df)REAL
```

This procedure delivers the probability that a random value drawn from the chi-squared distribution with df degrees of freedom will exceed x. For its use in testing goodness of fit of experimental results to theoretical hypotheses any standard textbook on Statistics should be consulted.

Fault display ILLEGAL PARAMETER OF PROC CHIPROB occurs if x < 0 or df < 1.



SNEDECOR'S F PROBABILITY DISTRIBUTION FUNCTION

```
PROC fprob = (REAL f, INT df1, df2)REAL
```

If f is the ratio of two sample variances with degrees of freedom $df1$ and $df2$, this procedure delivers the probability P that a value of f as large as that observed could have arisen by chance on the hypothesis that the two sample variances are independent estimates of the variance of the same normally distributed population. If P is small, the hypothesis may be considered to be correspondingly unlikely. The parameter f may have any non-negative value, and $df1$ and $df2$ any positive values. The algorithm used is exact subject to any machine round-off errors.

Fault display ILLEGAL PARAMETER OF PROC FPROB occurs if $f < 0.0$ or $df1 < 1$ or $df2 < 1$.

9.1 Use

The main use of the procedure is in assessing the significance of various factors in the so-called Analysis of Variance (a misnomer since it is actually sums of squares of variates and their degrees of freedom which are analysed into their component parts, not the resulting variances). A useful practical guide to the Analysis of Variance, without too much theory, is "Industrial Experimentation" by K A Brownlee, published by HMSO.

9.2 Example

As an example, the procedure could be used inside the procedure "continue" when using the library procedure polylimit (see Fitting 7) to test whether the last polynomial coefficient calculated is significant or not. The two variances concerned in this case are the variance $v1$ due to the coefficient, which has $df1 = 1$ degree of freedom and is therefore equal to R^2 , and the residual variance $v2$ which is equal to the sum of squares of the (possibly weighted) residuals in the y -fields of the array "points", divided by its degrees of freedom $df2$ which is equal to $(N - (n + 1))$, where N is the number of points and n is the current order of the polynomial (so that $(n + 1)$ is the number of coefficients being fitted).

The ratio $v1/v2$ then gives the value of f to be used as a parameter of $fprob$. The result P delivered gives the probability that a value of f as large as that observed could have arisen by chance if the true value of the last coefficient is zero. If the coefficient is not significant at the significance level chosen (eg if $P > 0.05$), at least one more coefficient should be tested before polylimit is terminated, in case the true polynomial consists of even (or odd) powers only. The maximum value of n used must not exceed $(N - 2)$, otherwise $df2$ would fall to zero.

If weights are used with polylimit, they should be inversely proportional to the rms errors (standard deviations) of the y -values of the various points, so that the weighted residuals will have a constant rms error independent of the x -values, as assumed by the f -test. For example, if the value y_i is actually the mean of N_i observations with the same value of x_i , and the original observations all have the same rms error, the weight z_i associated with y_i should be proportional to $\sqrt{N_i}$.

FISHER'S Z PROBABILITY DISTRIBUTION FUNCTION

```
PROC zprob = (REAL z, INT df1, df2)REAL
```

If $z = \frac{1}{2} \ln(f)$, where f is defined as in the procedure fprob (see Random 9), this procedure delivers the probability P that a value of z as large as that observed could have arisen by chance on the given hypothesis. The parameter z (which is equal to the difference of the natural logarithms of the two sample standard deviations concerned) may have any real value, and $df1$ and $df2$ any positive values.

Fault display ILLEGAL PARAMETER OF PROC ZPROB occurs if $df1 < 1$ or $df2 < 1$.

One advantage of the z -distribution over the f -distribution is its symmetry property, ie

$$zprob(z, df1, df2) = 1.0 - zprob(-z, df2, df1),$$

so that if the function were to be tabulated only positive values of z need be used, or for inverse tabulation which is more usual (ie tabulation of z for given values of P , $df1$ and $df2$), only values of P less than 0.5 need be used. In practice this is unimportant since values of P greater than 0.5 are seldom required.

Another advantage is that the range of values of z to be tabulated is more restricted than the range of corresponding values of f , and interpolation in the table is easier. Neither of these advantages apply if machine calculation is used instead of tables, so if there is no other reason for using zprob, users are recommended to use the procedure fprob instead.

STUDENT'S T PROBABILITY DISTRIBUTION FUNCTION

PROC tprob = (REAL t, INT df)REAL

This procedure delivers the probability P that a random value drawn from Student's t-distribution with df degrees of freedom will exceed ABS t in absolute value. The parameter t may have any real value, and df any positive value.

Fault display ILLEGAL PARAMETER OF PROC TPROB occurs if df < 1.

11.1 Use

The t-test has many uses in Statistics, perhaps the most common being to test whether any quantity (such as the mean of a sample) differs significantly from zero. The statistic t is defined as the ratio of the quantity to an estimate of its standard deviation based on df degrees of freedom, on the assumption that the quantity is normally distributed.

A generalisation is to test whether the means \bar{x} and \bar{y} of two samples x_i and y_j ($i = 1$ to m , $j = 1$ to n) are significantly different, or whether the samples could have been drawn from the same normal population. (For example, if some treatment had been applied to one sample and not to the other, this would test whether or not the treatment had any significant effect).

11.2 Method

Let S_x and S_y denote the total sums of squares of the variates x and y respectively. The sums of squares of the deviations of the two samples about their respective means are

$$S_1 = \sum (x_i - \bar{x})^2 = S_x - m\bar{x}^2 \quad \text{and}$$

$$S_2 = \sum (y_i - \bar{y})^2 = S_y - n\bar{y}^2 ,$$

the respective degrees of freedom being $(m - 1)$ and $(n - 1)$ since \bar{x} and \bar{y} have been calculated from the data. Estimates of the variances of the two samples are thus

$$S_1/(m - 1) \text{ and } S_2/(n - 1)$$

On the hypothesis that the two samples are from the same population, a revised estimate s^2 of the population variance is obtained by pooling the sums of squares and pooling the degrees of freedom, ie

$$s^2 = (S_1 + S_2)/(m + n - 2)$$

Estimates of the variances of the means \bar{x} and \bar{y} are therefore s^2/m and s^2/n respectively, and hence an estimate of the variance of the difference $(\bar{x} - \bar{y})$ is

$$s^2/m + s^2/n = s^2(m+n)/mn$$

since \bar{x} and \bar{y} are independent. We therefore substitute

$$t = \sqrt{mn/(m+n)} (\bar{x} - \bar{y})/s$$

as a parameter of tprob, with $df = (m+n-2)$. The result delivered gives the probability that a value of t as large as this in absolute value could have arisen by chance if the two samples were in fact drawn from the same normal population.

11.3 Equivalence of t-test and f-test

An alternative approach is to use the Analysis of Variance and the f-test (see Random 9). We regard the two samples as sub-samples of one large sample

z_k ($k = 1$ to $(m+n)$) with mean \bar{z} and total sum of squares of the sample values

$$S_z = S_x + S_y, \text{ where}$$

$$z_i = x_i \quad (i = 1 \text{ to } m) \quad \text{and} \quad z_{m+j} = y_j \quad (j = 1 \text{ to } n)$$

The total sum of squares (S_t) of deviations about the sample mean \bar{z} may be analysed into two independent parts, namely those due to differences within the sub-samples (S_w) and between the sub-samples (S_b). Using the relation $(m+n)\bar{z} = mx + ny$, we have

$$S_w = \sum (x_i - \bar{x})^2 + \sum (y_j - \bar{y})^2 = S_z - mx^2 - ny^2$$

$$S_b = m(\bar{x} - \bar{z})^2 + n(\bar{y} - \bar{z})^2 = mx^2 + ny^2 - (m+n)\bar{z}^2$$

$$S_t = (z_k - \bar{z})^2 = S_z - (m+n)\bar{z}^2$$

Clearly $S_w + S_b = S_t$. The degrees of freedom are also additive, being $(m+n-2)$, 1 and $(m+n-1)$ respectively.

We require to test whether the variance between the sub-samples is significantly greater than the variance within the sub-samples. The two variances to be compared are $v1 = S_b$ and $v2 = S_w/(m+n-2)$, and hence $f = (m+n-2)S_b/S_w$ with $df1 = 1$, $df2 = (m+n-2)$. Expressing \bar{z} in terms of \bar{x} and \bar{y} as above, it is easily shown that $f = t^2$ where t is the quantity previously calculated for the t-test (see 11.2). The procedures tprob and fprob therefore deliver the same probability P in view of the relation

$$\text{tprob}(t, df) \equiv \text{fprob}(t^2, 1, df)$$

The t-test and the f-test are therefore exactly equivalent in this case.

SIGNIFICANCE OF CORRELATION COEFFICIENTS

```
PROC rprob = (REAL r, INT df)REAL
```

If r is an estimate based on df degrees of freedom of a total or partial correlation coefficient between two variates, calculated from a sample of n points drawn from a multivariate gaussian distribution (ie a joint distribution of two or more normally distributed variates), this procedure delivers the probability that a value of r as large in absolute magnitude as that observed could have arisen by chance if the corresponding true correlation coefficient is zero in the population from which the sample is drawn. The absolute value of r never exceeds unity. For a total correlation coefficient, df is equal to $(n - 2)$ since two quantities (eg \bar{x} and \bar{y}) are estimated from the data. If r is a partial correlation coefficient, df must be reduced by a further 1 for each variate eliminated.

Fault display ILLEGAL PARAMETER OF PROC RPROB occurs if ABS $r > 1.0$ or $df < 1$.

12.1 Total correlation coefficients

The sample total correlation coefficient between two variates x and y is defined as

$$\begin{aligned} r_{xy} &= (\sum(x - \bar{x})(y - \bar{y})) / \sqrt{\sum(x - \bar{x})^2 \cdot \sum(y - \bar{y})^2} \\ &= (\sum xy - n\bar{x}\bar{y}) / \sqrt{(\sum x^2 - n\bar{x}^2)(\sum y^2 - n\bar{y}^2)} \\ &= (n\sum xy - (\sum x)(\sum y)) / \sqrt{(n\sum x^2 - (\sum x)^2)(n\sum y^2 - (\sum y)^2)} \end{aligned}$$

where the summations are made over the n points of the sample, and \bar{x} and \bar{y} are the sample means of the variates x and y respectively.

12.2 Efficiency and Accuracy

Care should be taken in evaluating formulae such as those above, which may involve subtractions of two large but nearly equal quantities. The last formula is the most efficient and most accurate since no divisions are involved in the numerator or denominator, especially if the data are supplied as integers which are stored exactly. This is easily done, since scaling any of the variates (or moving their origins) has no effect on correlation coefficients. However, it is advisable to store the data as real numbers to avoid possible overflow problems. There is normally no need to store the data in an array since any sums required (eg $\sum x$, $\sum y$, $\sum x^2$, $\sum y^2$, $\sum xy$) can be accumulated while the data are being read in.

12.3 Partial correlation coefficients

Part or all of the total correlation between two variates x and y may be due to their both being correlated with a further variate z , so that large values of z are mainly associated with large (or small) values of x and large (or small) values of y and contrariwise, causing an apparent correlation between x and y which does not really exist. This effect may be avoided by calculating the sample partial correlation coefficient between x and y with the effect of z eliminated, which is given by

$$r_{xy.z} = \frac{(r_{xy} - r_{xz}r_{yz})}{\sqrt{(1.0 - r_{xz}^2)(1.0 - r_{yz}^2)}}$$

To find the sample partial correlation coefficient between x and y when the effect of a further variate t is eliminated (as well as z), simply replace each of the correlation coefficients in the above formula by the corresponding partial correlation coefficient with t eliminated:-

$$r_{xy.zt} = \frac{(r_{xy.t} - r_{xz.t}r_{yz.t})}{\sqrt{(1.0 - r_{xz.t}^2)(1.0 - r_{yz.t}^2)}}$$

This process may be repeated indefinitely if required until the effects of all the variates except x and y have been eliminated (provided n is at least one more than the total number of variates, since df must be positive).

SORTING REAL NUMBERS IN CORE

PROC shellsort = (REF[]REAL vec)

Alters the array to which vec refers, re-ordering its elements in ascending numerical sequence, ie increasing with increasing index.

ORDERING ARBITRARY RECORDS IN CORE

```
PROC shellorder = (REF[ ]ANYMODE vec,  
                    PROC(REF ANYMODE, REF ANYMODE)BOOL swapped)
```

NOTE *This procedure cannot be called directly by the programmer, as the mode ANYMODE is not defined. Instead, a procedure with a different name, order (say), having parameters the same as shellorder but with the required mode in place of ANYMODE must be identified with "shellorder AS order", and can then be called. This is shown in the example on the next page.*

The procedure re-orders the elements of vec, using the ordering criterion supplied by swapped, with a minimum number of comparisons and interchanges. Swapped must inspect the values referred to by its parameters; if they are in the correct order it must exit delivering FALSE, otherwise it must order the two values correctly and deliver TRUE.

The method requires about $n \log_2 n$ comparisons and interchanges to order n values, uses no working space within shellorder, and is probably more efficient than any process written by the user.

The mode CHAR may not be used in place of ANYMODE, but structures can be ordered using a CHAR field as the key.

Examples are given on the next page.

Example To order an array of REALPAIR structures in ascending order of their y-fields. First declare a procedure similar to shellorder by the special construction:

```
PROC(REF[ ]REALPAIR, PROC(REF REALPAIR, REF REALPAIR)BOOL)order  
= shellorder AS order;
```

Secondly, declare an auxiliary procedure:

```
PROC aux = (REF REALPAIR u, v)BOOL:  
BEGIN REALPAIR w;  
    IF y OF u <= y OF v THEN FALSE  
    ELSE w:=u; u:=v; v:=w; TRUE  
    FI  
END;
```

Thirdly, assuming that the array of REALPAIR structures has been declared as

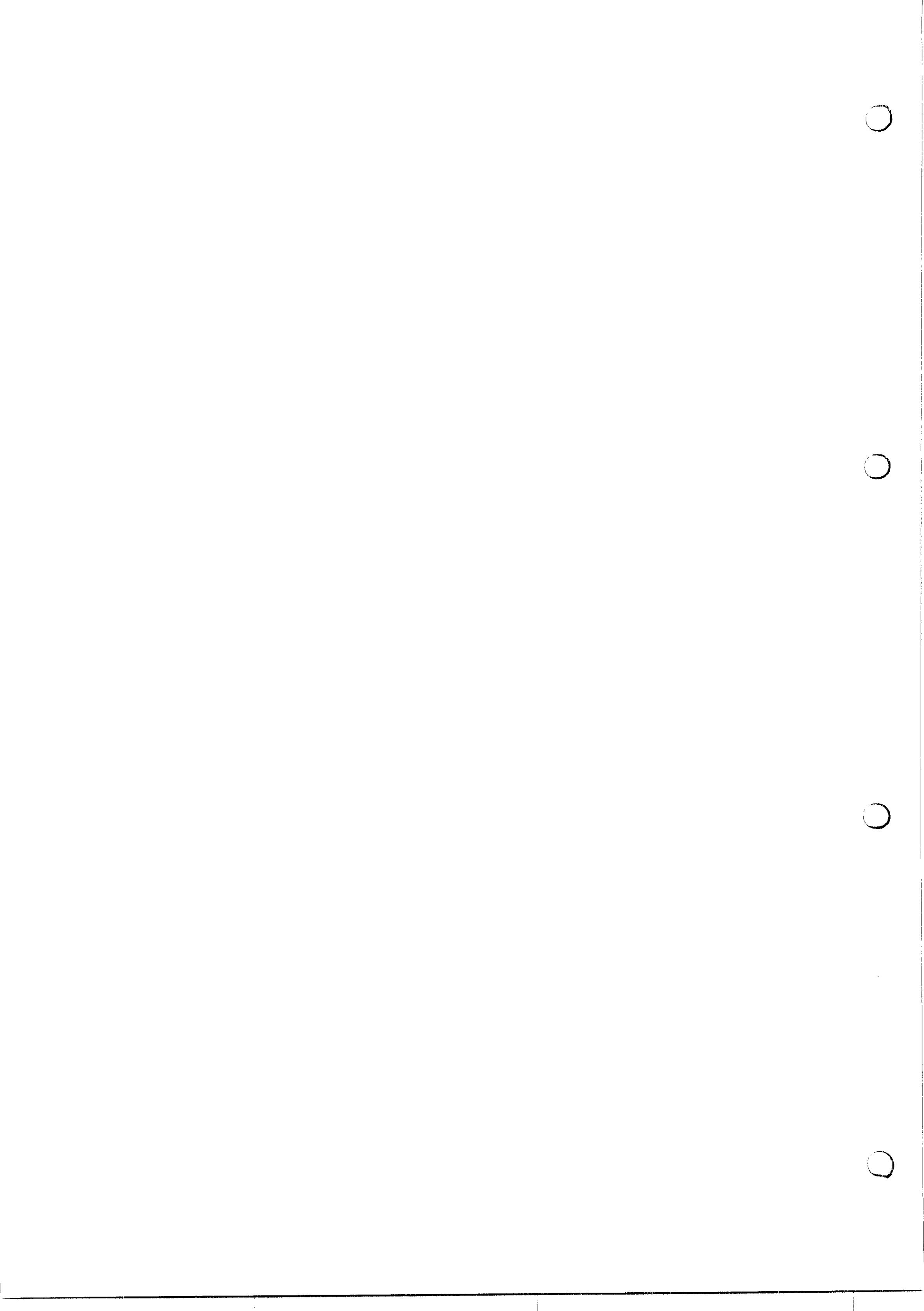
```
[1:n]REALPAIR array;
```

and assuming that this array has been set up by assignments, the ordering is effected by the call:

```
order(array, aux);
```

When the values to be ordered are large structures, it is more efficient to do an indirect ordering, using as ANYMODE a reference to the structure. For example,

```
[1:n]REALPAIR array;  
[1:n]REF REALPAIR refs;  
PROC(REF[ ]REF REALPAIR,  
     PROC(REF REF REALPAIR,  
          REF REF REALPAIR)BOOL) orderef  
= shellorder AS orderef;  
  
FOR i TO n DO refs[i] := array[i];  
PROC aux1 = (REF REF REALPAIR u, v)BOOL:  
BEGIN REF REALPAIR w;  
    IF y OF u <= y OF v THEN FALSE  
    ELSE w:=u; u:=v; v:=w; TRUE  
    FI  
END:  
----;  
orderef (refs, aux1)
```



NOTES ON THE USE OF FAST FOURIER TRANSFORMS

The so-called "Fast Fourier Transform" deals with the following pair of finite reciprocal transformations

$$(1) \quad u_r = \sum_s v_s \exp(2\pi i rs/N) \quad (\text{N terms})$$

$$(2) \quad Nv_s = \sum_r u_r \exp(-2\pi i rs/N) \quad (\text{N terms})$$

where both sums are taken over N successive values of the integer of summation. There are N complex numbers in the array u and in v , and the rapid computational method demands that N be chosen as a power of 2. The procedure "fousum" effectively computes u from v with s going from 0 to $N-1$, whilst "fourinvert" computes Nv from u with r going from 0 to $N-1$.

The above transform is more closely related to the Fourier Series than the Fourier Integral Transform. The Fourier Series for a complex periodic function $g(t)$ with period P may be written

$$(3) \quad g(t) = \sum_s A_s \exp(2\pi i st/P) \quad (-\infty \text{ to } +\infty)$$

where

$$(4) \quad P.A_s = \int_P g(t) \exp(-2\pi i st/P) dt \quad (\text{all } s)$$

By defining a unit of time \underline{t} , the series at the sample-points $r\underline{t}$ becomes

$$(5) \quad g(r\underline{t}) = \sum_s A_s \exp(2\pi i rs/N) \quad (-\infty \text{ to } +\infty)$$

where N is the period with respect to r, given by

$$(6) \quad N\underline{t} = P$$

If the series (5) is truncated to N terms and compared with (1), it will be seen that $g(r\underline{t})$ may be identified with u_r and A_s with v_s .

A useful work of reference is Trans IEEE (Audio Electroacoustics), June 1967, which contains several relevant papers.

ANALYSIS OF COMPLEX PERIODIC WAVEFORMS

The Fast Fourier Transform is immediately and exactly applicable to the spectral analysis of a periodic waveform which has no more than N lines in its spectrum (including those at negative frequencies). When there are only N terms, the text-book series (3) is the same as (1), so where we would mathematically use (4) to analyse the waveform, we can use (2) instead. However, it should be noted that there is an essential ambiguity in (2) concerning the physical interpretation of the integer s . In (4), the coefficient A_s is unequivocally associated with the frequency $s/(Nt)$ cycles per second. In (2), this may not be the case, as any multiple of N can be added to s without making any difference. This is due to the fact that u_r is a sampling of $g(t)$ at instants separated by t , between each of which the phase may have passed undetectably through any whole number of cycles. Recourse must be had to the known physical characteristics of the waveform before it was sampled, to clear up this ambiguity.

If there are more than N lines in the spectrum of $g(t)$, the exact equivalence of (1) and (3), and hence of (2) and (4), breaks down. There is no longer any equality between v_s and A_s . In principle, this difficulty can always be overcome by increasing N until convergence is obtained.

REAL PERIODIC WAVEFORMS

If $g(t)$ is purely real, the Fourier coefficients A_s have the symmetry property

$$A_{-s} = A_s^*$$

i.e. the line-spectrum is symmetrical in amplitude about zero frequency, and anti-symmetrical in phase. If the highest positive frequency component is at $s = h$, there are $2h + 1$ lines in the spectrum, and N should be chosen greater than $2h + 1$. The correspondence between v_s and A_s is then

A_{-h}	A_{-h+1}	\dots	A_{-1}	A_0	A_1	\dots	A_{h-1}	A_h
v_{N-h}	v_{N-h+1}	\dots	v_{N-1}	v_0	v_1	\dots	v_{h-1}	v_h

As the components v_0 to v_h contain all the information, only the bottom half of the output array from the library procedure need be used, though it should be remembered that both halves of the spectrum are necessary to ensure that all the cisoidal components combine to make a real waveform.

NOISE ANALYSIS

When a real noise fluctuation $g(t)$ is given in a limited interval P , its continuous power spectrum can only be estimated from the Fourier coefficients A_s . These are separated in frequency by $1/(Nt)$, which is the highest resolution obtainable from N points. The power associated with A_s is to be regarded as spread over a band of this width. The estimated spectral power density at frequency $s/(Nt)$ is thus

$$|A_s|^2 \frac{Nt}{N} = |Nv_s|^2 \frac{t}{N}$$

$$|\text{array element}|^2 \frac{t}{N}$$

where "array element" is an element of the array output by the procedure "fourinvert". Remember that the total power is divided equally between positive and negative frequencies.

Estimates of power density obtained in this way are subject to an error of 100%. It is therefore essential to repeat the analysis over several independent segments of a noise record, and average the results.

LIBRARY
Transforms
4

FAST FOURIER TRANSFORM (POSITIVE IMAGINARY EXPONENT)

PROC fousum = (REF[] COMPLEX z)

Let the number of complex elements in the actual array be denoted by N, which must be a power of 2. Whatever the actual bounds, let the elements be denoted by z_n with n running from 0 to N-1, i.e.

$z[\text{LWB } z]$ is here denoted by z_0

$z[\text{UPB } z]$ is denoted by z_{N-1}

The procedure alters the given array as though, for each element, it had performed the assignment

$$z_m := \sum_n z_n \exp(2\pi i nm/N)$$

with n and m both going from 0 to N-1. If the actual parameter does not refer to a number of elements which is an exact power of 2, the fault message is AD SIZE NOT POWER OF 2.

FAST FOURIER TRANSFORM (NEGATIVE IMAGINARY EXPONENT)

PROC fourinvert = (REF[] COMPLEX z)

The specification is identical with that of "fousum" above, except that

$$z_m := \sum_n z_n \exp(-2\pi i nm/N)$$

Successive application of fousum and fourinvert to a given array would have the effect of multiplying all the original elements by N. Note that z_m corresponds to Nv_s in equation (2) of the page Transforms 1.

When no phase information is to be used, for example when estimating a power spectrum, fousum and fourinvert are exactly equivalent.

INDEX OF LIBRARY NAMES

ABS Operators 4,5,7,10
addfrac Matrices 7
after OF numberstyle Input/Output 1.5
alter size Binary Transput 12.3
AND Operators 5,8
arccos Functions 1
arccosec Functions 1
arccot Functions 1
arcsec Functions 1
arcsin Functions 1
arctan Functions 1
ARG Operators 4
autograde Differential Equations 1
autoquad Quadrature 1

backspace Input/Output 1.4,3.1,7.2
before OF numberstyle Input/Output 1.5
BIN Operators 7
binary reader GEORGE 5.4.2
b of g Functions 7
buffered Binary Transput 4.1

cbrt Functions 1
cd Functions 12
cerf Functions 9
cerfc Functions 9
cexp Functions 1
ch Functions 1
charheader Binary Transput 6
charnumber Input/Output 3.2,5.3.5.1
charnumber g Binary Transput 5.3
charnumber p Binary Transput 5.3
CHARPUT Input/Output 2.2,2.3,7.1
 end OF Input/Output 3.5
 event OF Input/Output 5
 in OF Input/Output 6
 out OF Input/Output 6
charsleft Input/Output 3.2
chebsum Functions 13
chiprob Random 8
CLEAR Operators 6,9
clear format Input/Output 4.11.1
cln Functions 1
closeb Binary Transput 11,14.5
closec Input/Output 2.4
cn Functions 12
comerrfault Miscellaneous 5
common buffered Binary Transput 7.1
compellint Functions 6
COMPLEX Modes
CONJ Operators 4
converged Approximation 2
converged comp Approximation 2
copy format Input/Output 4.12.2

LIBRARY
Index
cos-fre

cos	Functions 1
cosec	Functions 1
cosech	Functions 1
cosh	Functions 1
cot	Functions 1
coth	Functions 1
cphi	Functions 9
cpower	Functions 10
cs	Functions 12
csqrt	Functions 1
CTB	Operators 11
CTLB	Operators 11
CTLLB	Operators 11
current charput	Input/Output 5.5.2
date	Miscellaneous 1
datechars	Miscellaneous 1
day	Miscellaneous 2
daychars	Miscellaneous 2
days	Miscellaneous 1
dc	Functions 12
direct	Binary Transput 4.2
disc	Binary Transput 15
disc details	Binary Transput 12.1
disc enquiry	Binary Transput 12.1
display	Miscellaneous 7
DIV	Operators 3
dn	Functions 12
ds	Functions 12
dump buffer	Binary Transput 8
ELEM	Operators 9,11
ellkratio	Functions 6
end OF charput	Input/Output 3.5
endfile	Binary Transput 3,4,4.1,14.5
ENTIER	Operators 4
equidif	Interpolation 2
equipol	Interpolation 2
erf	Functions 3
erfc	Functions 3
event OF charput	Input/Output 5
event OF get OF transit	Binary Transput 13
event OF put OF transit	Binary Transput 13
eventdata	Binary Transput 13.1
eventset	Binary Transput 13.1
exp	Functions 1
exp OF numberstyle	Input/Output 1.5
fault	Miscellaneous 7
fprob	Random 9
file printer	Input/Output 2.4
file punch	Input/Output 2.5.3
file reader	Input/Output 2.4
file size	Binary Transput 12.3
FLEXIBLE	Operators 6
format	Input/Output 4.11
fourinvert	Transforms 4
fousum	Transforms 4
free	Miscellaneous 4
fresnel	Functions 11

gamma
gap OF numberstyle
gaussian
gaussvar
geo3date
get
get OF transit
get bin
getc
get from
get next
g of b
grouped
groupsize

halfpi
hcf
headersize
hermitian qr
herpro

i0
i1
in
in OF charput
inf
infinity
input
int OF numberstyle
interdif
interpol
INTPAIR
intrandom
intrandvar
INTREAL
INTRIPLE
invellkratio
inverlate

j0
j1
jobname
jump g
jump p

k0
k1

lcm
LENG
linearfit
linenumber
linenumber g
linenumber p
ln
log10e
LWB

Functions 4
Input/Output 1.5
Random 4
Random 4
Miscellaneous 1
Input/Output 2,7.2
Binary Transput 7
Binary Transput 5
Equivalent to "get"
Binary Transput 5.1.1.
Equivalent to "get bin"
Functions 7
Binary Transput 6
Binary Transput 13.3

Constants
Functions 8
Binary Transput 13.3
Matrices 5
Matrices 6

Functions 2
Functions 2
Input/Output 4.11
Input/Output 6
Input/Output 4.1
Constants
Binary Transput 7,14.1.2
Input/Output 1.5
Interpolation 3
Interpolation 3
Modes
Random 3
Random 3
Modes
Modes
Functions 6
Interpolation 4

Functions 2
Functions 2
Constants
Binary Transput 6.1
Binary Transput 6.1

Functions 2
Functions 2

Functions 8
Operators 4
Fitting 8
Input/Output 3.2,5.3.5.1
Binary Transput 5.3,14.2.2
Binary Transput 5.3,14.2.2
Functions 1
Constants
Operators 6

LIBRARY
Index
mar-ove

mark Binary Transput 14.2.1
matmult Matrices 3
matmultcomp Matrices 3
MINUS Operators 3
minsumsq Fitting 11
monthchars Miscellaneous 2
multiquad Quadrature 2
multiquad comp Quadrature 2

nc Functions 12
nd Functions 12
negexp Random 5
negexpvar Random 5
newline Input/Output 1.4,3.1
newpage Input/Output 1.4,3.1
newton Fitting 9
next line Input/Output 5.3.5
next page Input/Output 5.3.5
next string Input/Output 2.5.5,5.3.5.1,7.3
normgauss Random 4
normintrand Random 3
normnegexp Random 5
normpoisson Random 7
normrand Random 2
normrayleigh Random 6
NOT Operators 5,8
now get bin Binary Transput 9
now get from Binary Transput 9
now jump g Binary Transput 9
now jump p Binary Transput 9
now mark Binary Transput 14.2.1
now put bin Binary Transput 9
now put to Binary Transput 9
now reset line Binary Transput 14.2.3
now setchar g Binary Transput 9
now setchar p Binary Transput 9
now setline g Binary Transput 9
now setline p Binary Transput 9
now skid Binary Transput 14.2.1
ns Functions 12
nul event Binary Transput 13.1
numberstyle Input/Output 1.5.1
NUMBERSTYLE Input/Output 1.5.4

obey command Miscellaneous 5
ODD Operators 4
off Miscellaneous 6
offset Binary Transput 7
on Miscellaneous 6
openarray Binary Transput 3
openb Binary Transput 4
openc Input/Output 2.4
OR Operators 5,8
out Input/Output 4.11
out OF charput Input/Output 6
outf Input/Output 4.1
output Binary Transput 7,14.1.2
OVER Operators 3
overlay Binary Transput 15

pagenumber
pagesize
pi
PLUS
poisson
poissonvar
polyfit
polylimit
polynomial
postmortem
print
put
put OF transit
put bin
putc
put line
put next
put to

random
random file
randvar
rayleigh
rayleighvar
read
REALPAIR
real roots
rename disc
rename tape
REPR
reset line
reset report
revive
rewind
root2
ROUND
rprob
save
sc
scapro
scapro comp
sd
sec
sech
seconds
seconds now
secsol
select both
select get
select put
serial file
SET
setchar g
setcharnumber
setchar p
set date
set datechars
set days
set geo3date

Input/Output 3.2,5.3.5.1
Input/Output 3.2
Constants
Operators 3
Random 7
Random 7
Fitting 6
Fitting 7
Functions 5
Miscellaneous 8
Input/Output 1
Input/Output 2
Binary Transput 7
Binary Transput 5
Equivalent to "put"
Binary Transput 14.1.1
Equivalent to "put bin"
Binary Transput 5.1.1

Random 2
Binary Transput 5.1
Random 2
Random 6
Random 6
Input/Output 1
Modes
Interpolation 5
Binary Transput 12.4
Binary Transput 14.4
Operators 10
Binary Transput 14.2.3
Miscellaneous 8
Miscellaneous 9
Binary Transput 14.2.3
Constants
Operators 4
Random 12
GEORGE 4.3.1
Functions 12
Matrices 6
Matrices 6
Functions 12
Functions 1
Functions 1
Miscellaneous 3
Miscellaneous 3
Fitting 10
Binary Transput 9
Binary Transput 9
Binary Transput 9
Binary Transput 5.2
Operators 9
Binary Transput 5.1
Input/Output 3.1,7.2
Binary Transput 5.1
Miscellaneous 1
Miscellaneous 1
Miscellaneous 1
Miscellaneous 1

LIBRARY Index set-tpr

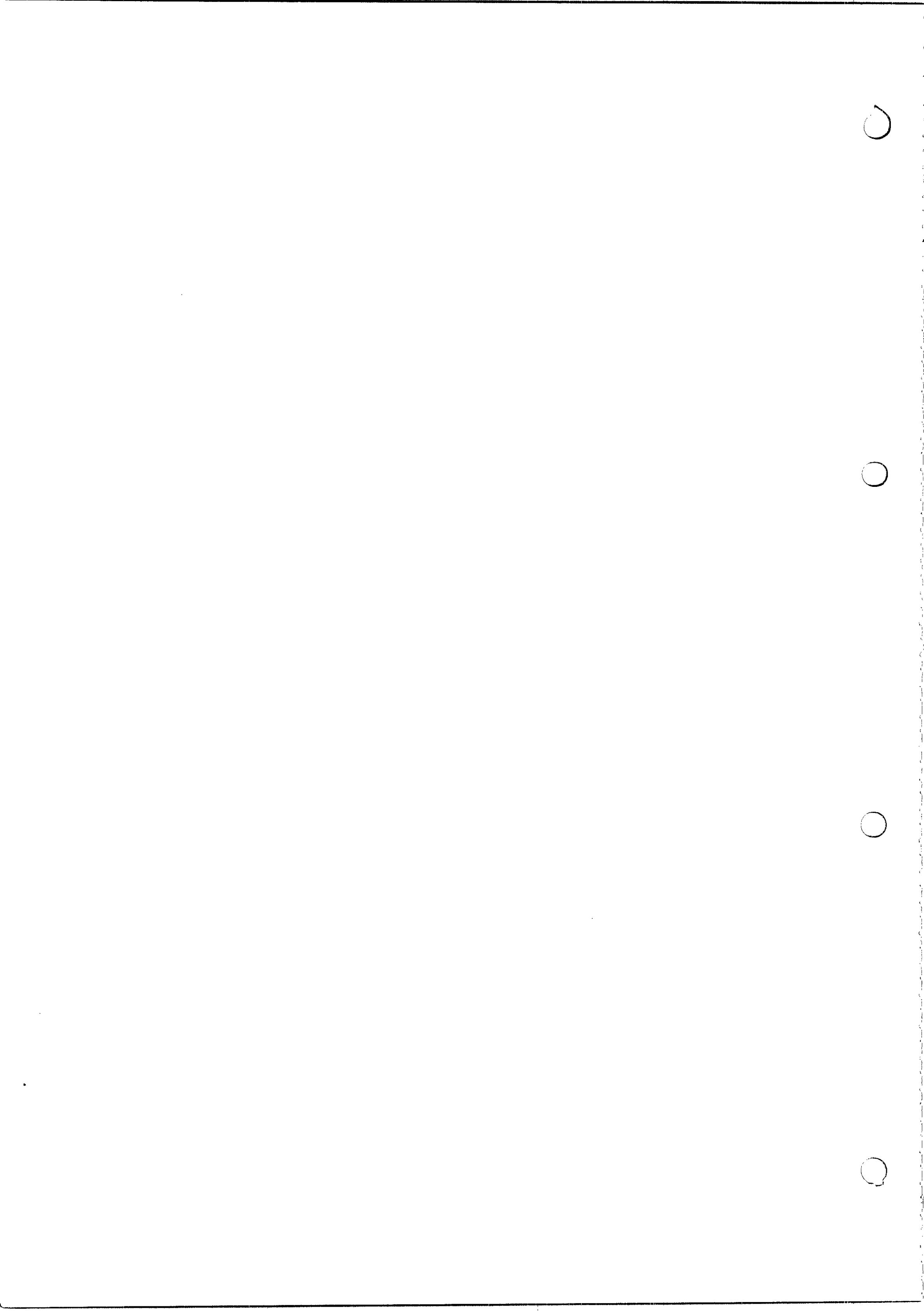
setline g	Binary Transput 5.1
setline p	Binary Transput 5.1
sh	Functions 1
shellorder	Sorting 2
shellsort	Sorting 1
SHORT	Operators 4
SIGN	Operators 4
sign OF numberstyle	Input/Output 1.5
simeq	Matrices 2
simeqcomp	Matrices 2
simeqcomp repeat	Matrices 2
simeq repeat	Matrices 2
sin	Functions 1
sinc	Functions 1
sinh	Functions 1
skid	Binary Transput 14.2.1
skip picture	Input/Output 4.11.2
SL	Operators 8
SLC	Operators 8
sn	Functions 12
space	Input/Output 1.4,3.1,7.2
spaces OF numberstyle	Input/Output 1.5
sqrt	Functions 1
SR	Operators 8
SRC	Operators 8
standaccuracy	Approximation 2
standin	Input/Output 2
standout	Input/Output 2
standput	Input/Output 3.4
standputc	Equivalent to "standput"
STRING	Modes
string lector	Input/Output 2.5.5
string printer	Input/Output 2.5.5,7.3
string punch	Input/Output 2.5.5
string reader	Input/Output 2.5.5
switch	Miscellaneous 6
switches	Miscellaneous 6
symmetric qr	Matrices 4
tan	Functions 1
tanh	Functions 1
tape	Binary Transput 14
tape details	Binary Transput 14.4
tape enquiry	Binary Transput 14.4
tapelinesize	Binary Transput 14.1
tape punch	Input/Output 2.5.4
tape reader	Input/Output 2.5.2
TH	Operators 9,11
th	Functions 1
timechars	Miscellaneous 3
timechars now	Miscellaneous 3
TIMES	Operators 3
tm number g	Binary Transput 14.2.2
tm number p	Binary Transput 14.2.2
tprob	Random 11

TRANSIT	Binary Transput	2.1
event OF get OF	Binary Transput	13
event OF put OF	Binary Transput	13
get OF	Binary Transput	7
put OF	Binary Transput	7
transpose	Matrices	8
twopi	Constants	

ungrouped Binary Transput 6
UPB Operators 6
user Constants

wide printer	Input/Output 2.5.1
wide reader	Input/Output 2.5.2
wordheader	Binary Transput 6
wrong	Miscellaneous 7

zprob Random 10



Rog J.

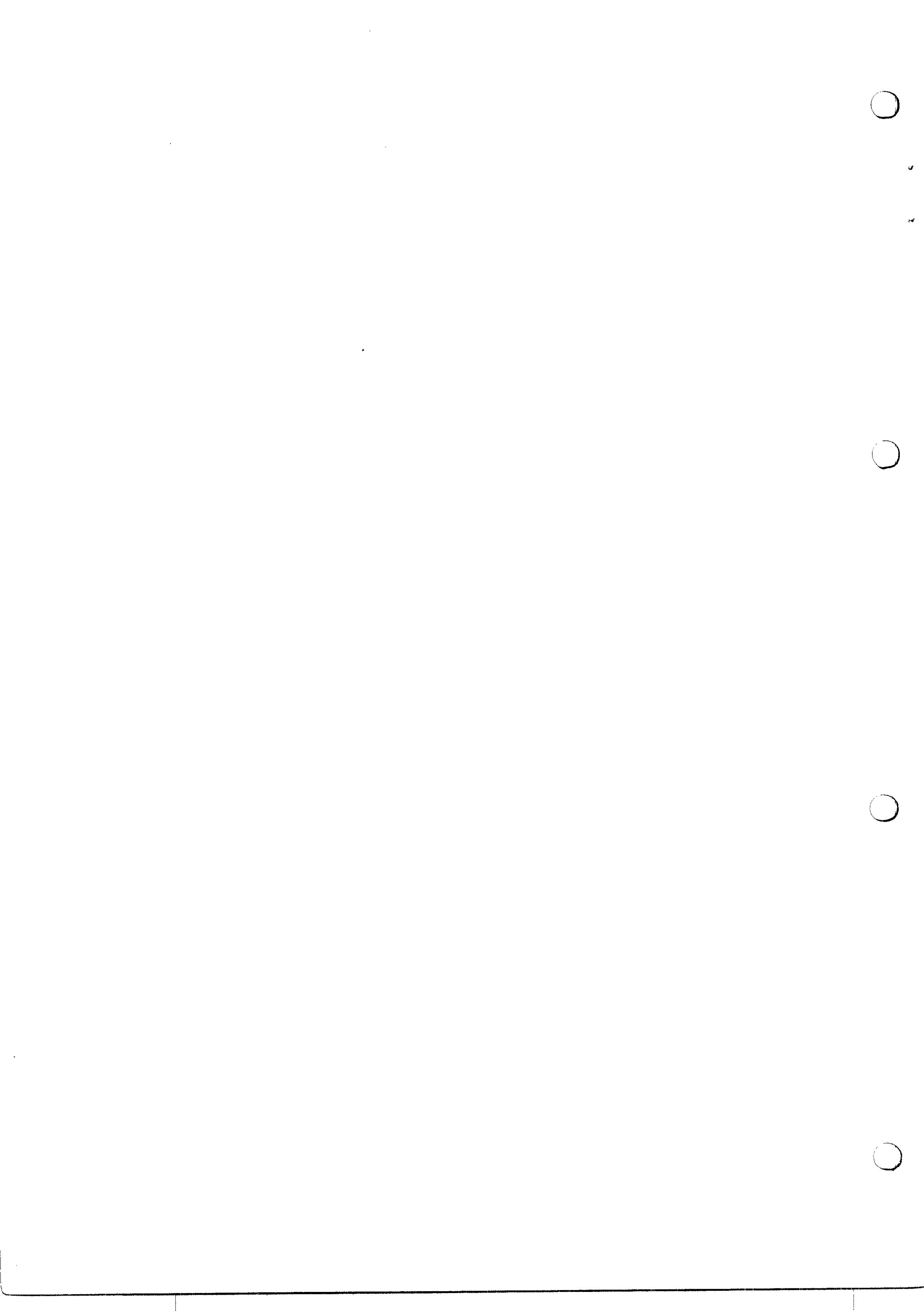
DOUBLE-PRECISION PACKAGE

The enclosed specifications for an Algol 68-R double-precision package are not officially part of the RRE Programmers' Manual as the segments concerned, although in the system library, are not invoked automatically but require special action by the user as described on the first page. However, because in several places they refer to other library specifications, it is thought that some users may wish to keep them with the rest of the library specifications. For this reason, and to facilitate any subsequent additions, they have been produced in the same loose-leaf form. All recipients will automatically receive any additions or amendments as issued.

Any requests for additional copies (giving name(s) and full address(es) of the person(s) requiring them), notification of any bugs, errors or obscurities, and any requests for additional procedures (which no undertaking can be given to fulfil) should be addressed directly to the undersigned.

S N Higgins
Computing and Software Research Division
Royal Radar Establishment
St Andrews Road
MALVERN
Worcestershire
WR14 3PS

Tel: Malvern (06845) 2733 Ext 3112.



1 AVAILABILITY

Double-precision real and complex arithmetic, transput, pseudo-denotations, standard operators and elementary mathematical functions are available by using the segment "longops" in the system library. This segment differs from all the other segments currently in the library in that it is the only one which keeps operators, apart from operators known to the compiler for which special measures can be taken. The result is that any segment using double-precision arithmetic must specifically mention the segment longops in its title, ie the title must be of the form

segname WITH longops FROM albumname

where albumname is the name of any album which does not contain the segment longops.

If any segments are required from an album, their names should appear after longops, and albumname should be the name of the album concerned.

If no segments are required from an album, but it is desired to put the compiled segment segname into an album, albumname should be the name of this album.

If neither of these two cases applies, albumname may be the name of any album.

Double-precision arithmetic should not be used for routine calculations owing to its comparative slowness, but only for special purposes for which its use is essential.

2 NUMBER FORMATS AND MODES

2.1 Real numbers

A double-precision real number is stored in a

MODE LREAL = STRUCT(INT p, q, r, s)

which occupies four words, the representation being the same as the ICL hardware representation for a double-precision floating-point number on the 1906A etc. If the number is zero, all four words are zero; otherwise the mantissa is standardised (ie bits 0 and 1 of the first word are different, bit 0 being the sign bit) and occupies the whole of the first word, bits 1 to 14 of the second word, bits 1 to 23 of the third word and bits 1 to 14 of the fourth word. The binary exponent increased by 256 occupies bits 15 to 23 of the second word, and bit 0 of the second, third and fourth word and bits 15 to 23 of the fourth word must be zero.

A LREAL number has 74 significant bits as opposed to 37 for an ordinary REAL number; for example, all integers in the range $\pm m$, where

$$m = 2^{74} = 188\ 89465\ 93147\ 85808\ 54784,$$

can be stored exactly as a LREAL. Any number of the form

$$n * 2^p$$

can also be stored exactly as a LREAL, where n is an integer in the range

$$-m \leq n < -m/2 \quad \text{or} \quad m/2 \leq n < m$$

and p is an integer in the range

$$-256 - 74 = -330 \leq p < 182 = +256 - 74.$$

The limits on the size of LREALS are for practical purposes the same as those on the size of REALS.

2.2 Complex numbers

A double-precision complex number is stored in a

MODE LCOMP = STRUCT(LREAL re, im)

occupying eight words. The real and imaginary parts must each be in the correct format of a LREAL as described in 2.1 above.

The modes LREAL and LCOMP are both kept in the segment longops and need not be declared by the user.

3 PSEUDO-DENOTATIONS

3.1 Format of a LREAL or LCOMP pseudo-denotation

There is no denotation for a LREAL or LCOMP which can be used in a program. However, a LREAL number may be written in the form of a REAL denotation enclosed in quotes and preceded by the operator LR, which converts the resulting []CHAR to a LREAL.

A LCOMP number may be written in the form of a pair of REAL denotations separated by ?, the whole being enclosed in quotes and preceded by the operator LC, which converts the resulting []CHAR to a LCOMP. If the imaginary part is zero, it (and the preceding ?) may be omitted; if the real part is zero, it may be omitted so that the []CHAR starts with ?.

With both LREAL and LCOMP numbers, the REAL denotations need not include a decimal point or exponent, so that they may look like INT denotations, except that they are not restricted to be less than 8388608. Each REAL denotation may be immediately preceded by a sign (after the " or ?) if required. Only the first 26 significant digits in the mantissa are taken account of, any further digits being treated as zeros. Each pseudo-denotation must contain at least one mantissa digit. Spaces and newlines in the []CHAR are ignored.

As there is no automatic widening coercion INT or REAL to LREAL, or COMPLEX to LCOMP, the appropriate pseudo-denotation must always be used when assigning a numerical constant to a LREAL or LCOMP variable, or when using it as a parameter of a procedure or as an operand requiring a LREAL or LCOMP value, or on the right-hand side of an identity declaration when naming a LREAL or LCOMP constant.

3.2 Accuracy of LREAL and LCOMP pseudo-denotations

Any number which can be stored exactly as a LREAL (see 2.1) will be stored exactly by writing its pseudo-denotation in the program. Even if it requires more than 26 significant decimal digits for its exact expression, only the first 26 need be written and this is sufficient to ensure its exact storage. Numbers such as LR "0.1", which cannot be stored exactly as a LREAL, will be stored to the full 74-bit accuracy (corresponding to just over 22 significant decimal digits).

With LCOMP pseudo-denotations, the above paragraph applies to the real and imaginary parts separately.

4 OPERATORS

4.1 Monadic operators

The following operators have the same meanings as the corresponding single-precision operators:-

<u>mode</u>	<u>operators</u>
OP (LREAL)LREAL	+ - ABS
OP (LREAL)INT	ROUND ENTIER SIGN
OP (LCOMP)LCOMP	+ - CONJ
OP (LCOMP)LREAL	ARG ABS

Additional operators are as follows:-

OP (LCOMP)LREAL	ABSQ
-----------------	------

Delivers the square of the modulus of its operand.

OP (REAL)LREAL	LR LENG (equivalent)
OP (INT)LREAL	LR

Delivers the LREAL corresponding to its REAL or INT operand.

OP (COMPLEX)LCOMP	LC LENG (equivalent)
OP (LREAL)LCOMP	LC
OP (REAL)LCOMP	LC
OP (INT)LCOMP	LC

Delivers the LCOMP corresponding to its COMPLEX, LREAL, REAL or INT operand.

OP (LREAL)REAL	SHORT
OP (LCOMP)COMPLEX	SHORT

Delivers the single-precision object corresponding to its LREAL or LCOMP operand, correctly rounded to 37 significant bits (unless this would cause it to overflow, when it is truncated instead).

The use of the operators LR and LC with []CHAR operands has already been dealt with in section 3.1. If the denotations in the []CHAR represent numbers which can be stored in the computer exactly, ie if they are either integers less than 8388608 or real numbers with less than 11 significant decimal digits which are equal to some integer times a positive or negative power of 2 (such as 0.75), it is more efficient to omit the pair of enclosing quotes after LR, or to replace those after LC by a pair of round brackets (in this case the real part must be present). The resulting INT, REAL or COMPLEX operand will then be converted directly to a LREAL or LCOMP.

4.2 Dyadic operators

The following operators have the same meanings and priorities as the corresponding single-precision operators:-

<u>mode</u>	<u>operators</u>
OP(LREAL, LREAL)LREAL	+ - * /
OP(LREAL, LCOMP)LCOMP	+ - * /
OP(LCOMP, LREAL)LCOMP	+ - * /
OP(LCOMP, LCOMP)LCOMP	+ - * /
OP(LREAL, LREAL)LCOMP	?
OP(LREAL, INT)LREAL	↑
OP(LCOMP, INT)LCOMP	↑ (similar to above)
OP(LREAL, LREAL)BOOL	< > <= >=

Additional operators are as follows:-

OP(REAL, REAL)LREAL	'*'
OP(REAL, COMPLEX)LCOMP	'*'
OP(COMPLEX, REAL)LCOMP	'*'

Delivers the exact product of its REAL or COMPLEX operands.

OP(COMPLEX, COMPLEX)LCOMP	'*'
---------------------------	-----

Delivers the LCOMP product of its COMPLEX operands. The multiplications of real and imaginary parts will be exact, but exactness may be lost in the subsequent addition and subtraction.

OP([]REAL, []REAL)LREAL	'*'
---------------------------	-----

Delivers the LREAL scalar product of two REAL vectors of the same size. The multiplication of corresponding array elements will be exact, but exactness may be lost in the additions.

OP([]LREAL, []LREAL)LREAL	*
-----------------------------	---

Delivers the LREAL scalar product of two LREAL vectors of the same size.

The LREAL or LCOMP results delivered by all the above operators, if not exact (where stated), should be correct to 74 significant bits.

It should be noted that there are no operators between single-precision quantities and double-precision quantities. The single-precision quantity must first be converted to double precision by using the operator LR or LC. If there is a choice, LR is to be preferred as the subsequent arithmetic operator routines are simpler and quicker.

'*' has the same priority (7) as *.

5 TRANSPUT

The transput of LREALs and LCOMPs cannot be done by the ordinary transput procedures put, get, print and read (except as 4 or 8 integers, which is not very convenient or illuminating); special procedures must be used which are detailed below. Formatted transput for LREALs and LCOMPs is not available (except as 4 or 8 integers).

The normal transput of LREALs and LCOMPs is governed by the fields of the variable

NUMBERSTYLE lnumberstyle;

lnumberstyle may be abbreviated to lns if preferred. The fields of lns have the same meaning as the corresponding fields of numberstyle, except for int OF lns which is not required for the transput of integers. If this is set to some positive integer n, a space will be inserted in the mantissa on output at intervals of n digits counting in both directions from the decimal point (if present), or from the right-hand end if not. If this facility is not required, int OF lns should be set to zero.

The default values of the fields of lns are as follows (for corresponding values and meanings of the fields of numberstyle see Input/Output 1.5.1 and 1.5.2. The additional settings of sign OF lns are also available corresponding to those given for sign OF numberstyle in Input/Output 1.5.3.1):-

<u>field of lns</u>	<u>standard setting</u>
spaces	2
sign	27
int	5
before	1
after	20
exp	2
gap	2

The above are also the fields of the constant "standlns" of mode NUMBERSTYLE. It should be noted that for the default values, 2 spaces are required to terminate a LREAL or LCOMP as opposed to 1 space for REALs, so that numbers output with internal spaces in the standard format can be read back in again.

With the default values, a LREAL number occupies 32 character positions on output (30 at the start of a line), and a LCOMP number occupies 64 positions (62 at the start of a line).

5.1 Input

A LREAL number to be input should be in the same form as a REAL number. Each LREAL is terminated in the same way as a REAL, ie by newline or newpage, by the last of g consecutive spaces where g = gap OF lns, or by any character which could not be part of the number. The reading position is left between the number and the terminator, as when inputting a REAL number using get or read.

A LCOMP number is input as a pair of LREAL numbers separated by ?. Each component is terminated as above.

Layout is ignored before the first digit or decimal point is read, and also after &, +, - or ? until the next digit or decimal point is read. For the default value of lns, single spaces may occur elsewhere. Only the first 26 significant digits of the mantissa (ie excluding any initial zeros) are taken account of, any further digits being treated as zeros.

Any number which can be stored exactly as a LREAL (see 2.1) can be input exactly. Even if it requires more than 26 significant decimal digits for its exact expression, only 26 digits need be input; this is sufficient to ensure that the number is stored exactly. Numbers such as 0.1 which cannot be stored exactly as a LREAL will be input with the full 74-bit accuracy (corresponding to just over 22 significant decimal digits).

For LCOMP numbers the above paragraph applies to the real and imaginary parts separately.

Separate procedures, which are listed below, must be used for inputting LREAL and LCOMP numbers. These cannot be used for inputting other objects.

5.1.1 LREAL input procedures

```
PROC lrget = (REF CHARPUT v, [ ]REF LREAL xx)
```

inputs the second parameter on the channel whose CHARPUT variable is v. The second parameter may be a single LREAL variable, or expression delivering a LREAL variable, or a collateral of such (mixed if desired), or a row or REF row of REF LREALS, but not a REF LREAL array of higher dimensions, nor a collateral of rows. The procedure lrget cannot be used to input a REF row of LREALS, for which the following procedure must be used:-

```
PROC lrgetrow = (REF CHARPUT v, REF[ ]LREAL xx)
```

```
PROC lrread = ([ ]REF LREAL xx)
```

```
PROC lrreadrow = (REF[ ]LREAL xx)
```

lrread(xx) and lrreadrow(xx) are exactly equivalent to lrget(standin, xx) and lrgetrow(standin, xx) respectively.

LIBRARY
Double Precision
5.1.2

5.1.2 LCOMP input procedures

```
PROC lcget = (REF CHARPUT v, [ ]REF LCOMP zz)
```

inputs the second parameter on the channel whose CHARPUT variable is v. The second parameter may be a single LCOMP variable, or expression delivering a LCOMP variable, or a collateral of such (mixed if desired), or a row or REF row of REF LCOMPs, but not a REF LCOMP array of higher dimensions, nor a collateral of rows. The procedure lcget cannot be used to input a REF row of LCOMPs, for which the following procedure must be used:-

```
PROC lcgetrow = (REF CHARPUT v, REF[ ]LCOMP zz)
```

```
PROC lcread = ([ ]REF LCOMP zz)
```

```
PROC lcreadrow = (REF[ ]LCOMP zz)
```

lcread(zz) and lcreadrow(zz) are exactly equivalent to lcget(standin, zz) and lcgetrow(standin, zz) respectively.

5.2 Output

A LREAL number is output in the same form as a REAL number, except that extra spaces may be inserted in the mantissa at intervals determined by int OF lns (default value 5). The total number of mantissa digits output (ie before OF lns + after OF lns) must not exceed 26. The number as stored will be output correctly rounded to the number of figures requested. If there is insufficient room to output the number on the current line, a newline is taken automatically before the number is output.

A LCOMP number is output as a pair of LREAL numbers, except that the imaginary part is preceded by a space and ? instead of by s spaces, where s = spaces OF lns.

Separate procedures, which are listed below, must be used for outputting LREAL and LCOMP numbers. They cannot be used for outputting other objects or for obeying procedures such as newline.

5.2.1 LREAL output procedures

PROC lrput = (REF CHARPUT v, []LREAL xx)

outputs the second parameter on the channel whose CHARPUT variable is v. The second parameter may be either a single LREAL constant, variable or expression, or a collateral of such (mixed if desired), or a LREAL row constant (defined by an identity declaration) or variable, but not a LREAL array of higher dimensions, nor a collateral of rows.

PROC lrprint = ([]LREAL xx)

lrprint(xx) is exactly equivalent to lrput(standout, xx).

PROC lrstandput = (REF CHARPUT v, []LREAL xx)

Specification is as for lrput but output is in the standard default style (see Double Precision 5). lrstandput may be used if lns has been altered and it is desired temporarily to output some LREALs in the standard style without altering lns.

5.2.2 LCOMP output procedures

PROC lcput = (REF CHARPUT v, []LCOMP zz)

outputs the second parameter on the channel whose CHARPUT variable is v. The second parameter may be either a single LCOMP constant, variable or expression, or a collateral of such (mixed if desired), or a LCOMP row constant (defined by an identity declaration) or variable, but not a LCOMP array of higher dimensions, nor a collateral of rows.

PROC lcprint = ([]LCOMP zz)

lcprint(zz) is exactly equivalent to lcput(standout, zz).

PROC lcstandput = (REF CHARPUT v, []LCOMP zz)

Specification is as for lcput but output is in the standard default style (see Double Precision 5). lcstandput may be used if lns has been altered and it is desired temporarily to output some LCOMPs in the standard style without altering lns.

6 ARITHMETIC PROCEDURES

Double-precision versions of all the elementary function procedures listed in Functions 1 are available, specifications being similar to those of the corresponding single-precision procedures obtained by omitting the prefix "lr" (for LREAL procedures) or "l" (for LCOMP procedures). The procedures are as follows:-

```
PROC(LREAL)LREAL : lrsqrt, lrcbrt, lrexp, lrln,  
                    lrsin, lrcos, lrtan, lrcot, lrsec, lrcosec,  
                    lrsinh, lrcosh, lrtanh, lrcoth, lrsech, lrcosech,  
                    lrarcsin, lrarccos, lrarctan,  
                    lrarccot, lrarcsec, lrarccosec, lrsinc.
```

```
PROC(LCOMP)LCOMP : lcsqrt, lcexp, lcln.
```

The following procedures are also available:-

```
PROC(LREAL, [ ]LREAL)LREAL : lrpolynomial.  
PROC(LCOMP, [ ]LCOMP)LCOMP : lcpolynomial.
```

Specifications are similar to that of polynomial (see Functions 5).

7 CONSTANTS

The following double-precision constants are available without declaration:-

LREAL	lrzero	= LR 0,
	lrone	= LR 1,
	lrhalf	= LR 0.5,
	lrten	= LR 10,
	lrpi	= LR "3.14159 26535 89793 23846 26434",
	lrhalfpi	= LR "1.57079 63267 94896 61923 13217",
	lrtwopi	= LR "6.28318 53071 79586 47692 52868",
	lrinfinity	= LR "5.78960 44618 65809 77117 82422 & 76";
LCOMP	lczero	= LC 0,
	lcone	= LC 1,
	lchalf	= LC 0.5;

lrinfinity is the largest LREAL number which can be stored.

Many other double-precision constants can be supplied to individual users if required.

8 FAULT DISPLAYS

The following fault displays may occur during a double-precision procedure or operator call. The name and position in the program of the particular procedure or operator call concerned may be obtained from the diagnostics.

ILLEGAL PARAMETER FORMAT - 1) These occur if the format of one or both
 ILLEGAL PARAMETER FORMAT - 2) LREAL or LCOMP parameters of the procedure
 ILLEGAL PARAMETER FORMAT - 3) or operator does not conform to that
 ILLEGAL PARAMETER FORMAT - A) described in 2.1. The first means that
 the first (or only) parameter is illegal,
 the second means that the second parameter is illegal, while the third
 means that both parameters are illegal. The fourth means that at least one
 element of an array parameter is illegal; it may occur during output
 procedures if the format of any one of the numbers to be output is illegal,
 or during a call of lrpolynomial or lcpolynomial, or of the operator * with
 []LREAL operands. The most likely cause is that the parameter concerned
 has not been properly initialised.

OVERFLOWED AT CALL

occurs if overflow was already set when
 the procedure or operator was called.

The error will probably be in some previous single-precision calculation.

OVERFLOW SET

occurs if overflow is set during the
 procedure or operator call. This may be
 caused by (eg) division by zero, or inputting a number larger than
 lrinfinity, or calculating lrten + 80, or lrexp(LR 200).

ILLEGAL PARAMETER VALUE

occurs if the parameter of a particular
 procedure is illegal, eg if lrsqrt or lrln
 is called with a negative parameter, or if lrarcsin is called with a
 parameter greater than unity.

ILLEGAL CHARACTER

occurs during input procedures if, for
 each number, the first non-layout
 character read could not be the start of a number, or if an illegal
 character occurs after "+", "-", "&" or "?". It also occurs during calls
 of the operators LR or LC with a []CHAR operand if an illegal character
 occurs anywhere in the row.

NUMBER FORMAT ERROR

occurs during output procedures if
 exp OF lns = 0 and the mantissa is too
 large for the number of digits assigned to before OF lns, or if the
 exponent is too large for the number of digits assigned to exp OF lns, or
 if more than 26 mantissa digits are requested, or if the number is negative
 and no sign is requested (ie the last two octal digits of sign OF lns are
 zero). In these cases the number is output (on a new line) in the standard
 default format before the fault display.

ARRAY SIZES INCOMPATIBLE

occurs during a call of the operator *
 with []LREAL operands, or of the operator
 '*' with []REAL operands, if the two rows are of different sizes.

LIBRARY
Double Precision
9

9 MATRIX MULTIPLICATION

```
PROC lrmatmult = (REF[ , ]LREAL a, b, c)
PROC lcmatmult = (REF[ , ]LCOMP a, b, c)
```

These procedures are for the multiplication of double-precision real and complex matrices respectively. If the matrices to which a, b and c refer are compatible in size, the procedure assigns the product of a and b to c, otherwise fault display MATRICES NOT COMPATIBLE occurs.

10 TRANSPOSING A MATRIX

```
PROC lrtranspose = (REF[ , ]LREAL a) REF[ , ]LREAL
PROC lctranspose = (REF[ , ]LCOMP a) REF[ , ]LCOMP
```

Each of these procedures delivers a reference to the transpose of the LREAL or LCOMP matrix to which "a" refers without actually moving or copying any of its elements. This reference should be identified with a new identifier (atrans, say) for subsequent use. If the original matrix (aorig, say) is still required, the actual parameter should not be the simple identifier "aorig", but a slice reference to the whole matrix, ie aorig[,], which has the same bounds. Thus, after obeying the piece of program

```
INT p, q, r, s; read((p, q, r, s));
[p:q, r:s]LREAL aorig;                                ) modify suitably
REF[ , ]LREAL atrans = lrtranspose(aorig[ , ])      ) for lctranspose
```

the bounds of atrans will be [r:s, p:q], and atrans[j, i] will refer to the same matrix element as aorig[i, j] for all values of i and j within the bounds.

This means that the original and the transposed matrices are both available simultaneously without any matrix elements having been copied, and they will remain transposes of each other however either of them is altered. The two identifiers do not refer to two different sets of matrix elements but to the same set of elements in different orders. It is not necessary for the matrix elements to be initialised before the procedure is called.

11 INVERTING A MATRIX

```
PROC lrinvert = (REF[ , ]LREAL a) REAL
PROC lcinvert = (REF[ , ]LCOMP a) REAL
```

Each of these procedures replaces the LREAL or LCOMP matrix to which the parameter refers by its inverse. If the original matrix is still required it should be copied first. The array a may have any bounds provided its two dimensions are equal. Fault display MATRIX NOT SQUARE occurs if this is not the case.

The rows and columns of a are first scaled by powers of 2 so that the largest element (or the largest real or imaginary part of an element in the case of lcinvert) in each row and in each column lies between 0.5 and 1.0 in absolute value. The resulting matrix is then inverted, and the columns and rows of the inverse matrix are scaled by the same powers of 2 as the rows and columns of the original matrix. This is to avoid overflow in case (eg) all the elements of a 10th order matrix were of the order 1&10, and to avoid the determinant becoming zero if all the elements were of the order 1&-10. Scaling the whole matrix (or any row or column) should not affect the relative accuracy of any element of the inverse matrix.

If the inversion is successful, the procedures each deliver an estimate of the largest absolute error in any element of the inverse matrix, divided by the largest absolute element, calculated before the rescaling takes place. (For lcinvert these refer to the real and imaginary parts of the errors and elements respectively). For most matrices this is normally less than 1&-22. For very badly conditioned matrices which are nearly singular to single precision (eg a 9th order Hilbert matrix) it may increase to 1&-16. For even worse conditioned matrices ~~are~~ singular to single precision (after scaling), or so nearly singular that the iterative process used does not converge (eg a 10th order Hilbert matrix), a negative result is delivered. In this case the contents of a will be meaningless.

If n is the order of the matrix, lrinvert and lcinvert require working space of the order of $12n^2$ and $24n^2$ words respectively, in addition to the space required for the matrix a. The time required varies roughly as the cube of n, lcinvert taking roughly four times as long as lrinvert (although the time increases for badly conditioned matrices).

12 ADDITION, SUBTRACTION & NEGATION OF VECTORS & MATRICES

```

PROC lrvecadd = (REF[ ]LREAL a, b, c)
PROC lcvecadd = (REF[ ]LCOMP a, b, c)
PROC lrmatadd = (REF[ , ]LREAL a, b, c)
PROC lcmatadd = (REF[ , ]LCOMP a, b, c)

```

Each of these procedures assigns to each element of c the corresponding element of a plus the corresponding element of b.

```

PROC lrvecsub = (REF[ ]LREAL a, b, c)
PROC lcvecsub = (REF[ ]LCOMP a, b, c)
PROC lrmatsub = (REF[ , ]LREAL a, b, c)
PROC lcmatsub = (REF[ , ]LCOMP a, b, c)

```

Each of these procedures assigns to each element of c the corresponding element of a minus the corresponding element of b.

```

PROC lrvecneg = (REF[ ]LREAL a, b)
PROC lcvecneg = (REF[ ]LCOMP a, b)
PROC lrmatneg = (REF[ , ]LREAL a, b)
PROC lcmatneg = (REF[ , ]LCOMP a, b)

```

Each of these procedures assigns to each element of b the negative of the corresponding element of a.

In each of the above twelve procedures the vectors or matrices a, b, c may have any bounds provided they all have the same dimensions. Fault display ARRAY SIZES INCOMPATIBLE occurs if this is not the case.

Unlike lrmatmult and lcmatmult where c must not refer to the same matrix as a or b (although a and b may refer to the same matrix), in these procedures a, b and c may all refer to the same vector or matrix. Thus, for example,

```

lrmatadd(a, a, a) would double each element of a,
lrmatsub(a, a, a) would clear a, and
lrmatneg(a, a) would negate each element of a.

```

(N.B. The second example would only work provided a had already been initialised with some valid LREALs. In practice the operator CLEAR would be used for this purpose).

13 INTEGER ARITHMETIC USING LREALS

```
PROC lrentier = (LREAL xx) LREAL
PROC lrround = (LREAL xx) LREAL
```

These procedures deliver respectively the largest (most positive) integer not greater than xx, and the nearest integer to xx (rounding up in critical cases), each as a LREAL. They give the correct answer for any value of xx, unlike the operators ENTIER and ROUND which may overflow for operands greater than 8388607 in absolute value. Integer division for large integers (up to 2^{74} in absolute value) can be implemented using these procedures if required.

```
PROC lrhcf = ([ ]LREAL a) LREAL
PROC lrlcm = ([ ]LREAL a) LREAL
```

These procedures deliver respectively the highest common factor and the lowest common (positive) multiple of the numbers contained in the array a, each as a LREAL. The numbers must be integers less than 2^{74} in absolute value, although in the form of LREALs. Fault display ILLEGAL PARAMETER VALUE occurs if this is not the case.

Although any number is a factor of zero, lrhcf delivers lrzero if all the elements of the array are zero. The procedure lrlcm delivers lrzero if any element of the array is zero.

If the lcm of the numbers in the array is not less than 2^{74} , lrlcm delivers the value -lrone, as it might not be possible to give the correct value as a LREAL even if it did not overflow.

Note that in both procedures, if the numbers in the array a are given as a collateral, an extra pair of brackets is required around the collateral as with the procedure "print". This is different from the single-precision procedures hcf and lcm (see Functions 8) which each have two parameters and only operate on two numbers.

