C64

ourne House

SOFTWARE

PROJECTS

C 6 4

RUDOLF SMIT

COMMODORE 64

SOFTWARE

PROJECTS

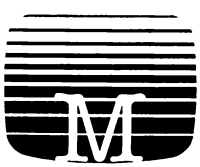
BY RUDOLF SMIT

COMMODORE 64

SOFTWARE

PROJECTS

BY RUDOLF SMIT



MELBOURNE HOUSE

Published in the United States of America by:
Melbourne House Software Inc.,
347 Reedwood Drive,
Nashville TN 37217.

Published in the United Kingdom by:
Melbourne House (Publishers) Ltd.,
Melbourne House
Church Yard,
Tring, Hertfordshire HP23 5LU.
ISBN 0 86161 146 2

Published in Australia by:
Melbourne House (Australia) Pty. Ltd.,
70 Park Street,
South Melbourne, Victoria, 3205.

Printed in Hong Kong by Colorcraft Ltd.
1st Edition

D C B A 9 8 7 6 5 4 3 2 1 0

CONTENTS

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7

Appendix I

Appendix II

Appendix III

INTRODUCTION

This book is for you if:

1. you have only recently acquired your Commodore 64,
2. you have worked through the user guide and gained some proficiency in cursor movements,
3. you have become acquainted with most of the Commodore BASIC keywords,
4. you are nevertheless very much a beginner and need clear guidance in the development of your own programs.

This book helps you to develop your programming skills, not by presenting entirely developed programs that have only to be keyed in, but by documenting the main 'flow' of programs. It will then be up to you to write the remaining procedures, using the techniques and hints in this book to guide you. In the rare event that you are unable to find a good solution to a programming problem, an appendix is provided with keys to solutions to which you can refer.

All programming projects included are ready for writing in accordance with the principles of the top-down approach in structured programming, a philosophy which has been widely accepted in the world of professional computing and is gaining a foothold, at last, among programmers in the BASIC language. Chapter 1 will deal in detail with the top-down approach.

Should you be a total newcomer to programming, we recommend that you use this book in conjunction with 'DISCOVER YOUR COMMODORE 64', also published by Melbourne House.

CHAPTER 1
The Right Way to Good Programming

1.1 The joy of programming

The 'Eighties' will probably enter history as the decade in which millions of people took up home computer programming and found it a fascinating pastime. And rightly so because, since programming requires both logical and creative thinking, it can be a highly stimulating mental exercise.

Programming is like chess — there are thousands of different routes to the ultimate goal of checkmate. Chess requires not only logic but the creative ability to visualise a multitude of solutions to the same problem. Similar processes are involved in successful programming. Assign a programming task to seven programmers and most likely they will come up with seven different solutions to the problem, each of them effective insofar as they do the job required of them. But this is probably about the only thing they have in common. There are many ways of doing a job — good, bad and indifferent. The science of computing is scarcely forty years old and programming techniques are still evolving, so it will be some time yet before there are any established standards for program construction.

Nevertheless, more is required of a program than merely delivering a correct result: it should satisfy certain design criteria as well. These criteria are summarised in the rules of the top-down approach in structured programming.

1.2 The basis of the projects in this book

All the projects in this book have two things in common — they're presented in BASIC (the most popular computer language of the pres-

ent time) and, more importantly, programming concepts are based on the principles of the top-down approach in structured programming.

This programming philosophy was first elaborated by a Dutch computer scientist, professor Dijkstra of Eindhoven in the Netherlands, and it has since found wide acceptance in the world of professional computing.

It is now gaining a foothold with programmers of personal computers. And not before time, either, because although BASIC may be the most popular computer language, its popularity is confined mainly to the users of personal and home computers. BASIC is ignored by many computer professionals who tend to look with some scorn on it and its users. The reason is that BASIC lends itself to what is regarded as 'unstructured' programming, and hence to what are seen as 'bad programming habits'. Moreover, tertiary institutes for computer training are often less than enthusiastic about students who have developed proficiency in BASIC, because some 'unlearning' is required before these students can enter the field of professional programming.

Hopefully this will change now that structured programming is being applied to BASIC. By adopting this particular programming philosophy, BASIC has come of age. One of the aims of this book is to contribute to the wider acceptance of the top-down approach in structured programming among users of BASIC.

1.3 Structured programming versus the trial-and-error approach

Let us look at two examples of BASIC programs. The first (A) is designed according to the principles of structured programming. The second (B) highlights the trial-and-error approach.

Example A

```
100 REM *** CONVERTER HRS.MNS.SCS TO DECIMAL HOURS ***
105 :
110 REM *** MAINLINE ***************************************************
115 :
120 GOSUB 200      :REM — TO INPUT ROUTINE
130 GOSUB 300      :REM — TO PROCESSING ROUTINE
140 GOSUB 400      :REM — TO CALCULATION ROUTINE
150 GOSUB 500      :REM — TO OUTPUT ROUTINE
160 END            :REM — PROGRAM ENDS HERE
```

```
17Ø :
2ØØ REM *** INPUT ROUTINE ********************************************
2Ø5 :
21Ø PRINT "♡"                    :REM — CLEAR SCREEN
215 :
22Ø PRINT"ENTER TIME
(FORMAT: HH.MM.SS)"              :REM — SCREEN PROMPT
23Ø INPUT HR$                    :REM — DATA IS PLACED IN HR$
24Ø RETURN                       :REM — RETURN TO MAINLINE
25Ø :
3ØØ REM *** PROCESSING ROUTINE ***********************************
3Ø5 :
31Ø SS$ = RIGHT$(HR$,2)          :REM — RETRIEVE SECONDS FROM
                                  HR$
32Ø MM$ = MID$(HR$,4,2)          :REM — RETRIEVE MINUTES FROM
                                  HR$
33Ø HH$ = LEFT$(HR$,2)           :REM — RETRIEVE HOURS FROM
                                  HR$
335 :
34Ø SS  = VAL(SS$)               :REM — MAKE STRING VALUE SS$
                                  NUMERICAL
35Ø MM  = VAL(MM$)               :REM — MAKE STRING VALUE MM$
                                  NUMERICAL
36Ø HH  = VAL(HH$)               :REM — MAKE STRING VALUE HH$
                                  NUMERICAL
37Ø RETURN                       :REM — RETURN TO MAINLINE
38Ø :
4ØØ REM *** CALCULATION ROUTINE  *********************************
4Ø5 :
41Ø SS = SS / 6Ø                 :REM — DIVIDE SECONDS BY 6Ø
42Ø MM = MM + SS                 :REM — ADD DECIMAL SECONDS
                                  TO MINUTES
43Ø MM = MM / 6Ø                 :REM — DIVIDE MINUTES BY 6Ø
44Ø HH = HH + MM                 :REM — ADD DECIMAL MINUTES TO
                                  HOURS
45Ø RETURN                       :REM — RETURN TO MAINLINE
46Ø :
5ØØ REM *** OUTPUT ROUTINE ***************************************
5Ø5 :
51Ø PRINT                        :REM — PRINT A BLANK LINE
52Ø PRINT "IN DECIMAL
    HOURS :";HH                  :REM — DISPLAY RESULT
53Ø RETURN                       :REM — RETURN TO MAINLINE
```

NOTE: 22Ø and 52Ø are ONE line each!

Example B

```
316Ø  Z=P1:GOSUB134Ø:GOSUB329Ø:P=7:PA(P)=PA
317Ø  GOSUB13ØØ:Q=P+12
318Ø  IFN=5THENG=G+18Ø
319Ø  IFG>36ØTHENG=G-36Ø
32ØØ  L=G:PL(Q)=L:GOSUB144Ø:GOSUB4Ø1Ø
321Ø  PRINT"□□□□":PRINTTAB(15)PO$;SI$
322Ø  N=Ø:Z=P2:GOSUB134Ø:GOSUB329Ø:P=8:PA(P)=PA
323Ø  GOSUB13ØØ:Q=P+12
324Ø  IFN=5THENG=G+18Ø
325Ø  IFG>36ØTHENG=G-36Ø
326Ø  L=G:PL(Q)=L:GOSUB144Ø:PRINT:PRINT:" □□□□ ":
      PRINTTAB(27)PO$;SI$:GOTO29ØØ
327Ø  G=G+18Ø:IFG>36ØTHENG=G-36Ø
328Ø  N=5:GOTO33ØØ
329Ø  Z=Z/FNCO(EC):GOSUB136Ø
33ØØ  SD=FNAS(FNSI(G)*LF)+9Ø:MD=FNEQ(G-RB):PA=MD/SD
      :IFABS(PA)>1THEN327Ø
331Ø  IFN=5THENPA=PA+5
332Ø  IFABS(PA)>1THENPA=PA-5
```

A glance at these two examples reveals it all. Example A is readable and neatly written, though you may not follow its argument just yet. The second example, on the other hand, deserves only one verdict: it is an indecipherable hotch-potch. (I am at liberty to be quite disrespectful about this program, because it is part of one of my own early attempts to master BASIC.)

These days there are many books about programming in BASIC and many popular computer magazines contain BASIC programs as well. A close study reveals that the majority suffer from the same ailments as B. Even trained programmers would have trouble figuring out how such programs work. It is not unusual for their authors to experience great difficulty in deciphering their own products after they have finished them. This is all due to poor program design. They very often have virtually no structure and are poorly documented.

1.4 Trial-and-error, the blessing and the curse of BASIC

Programs of the type shown in example B are the result of the trial-and-error approach that is so common in BASIC programming. BASIC is an

acronym for Beginners All-purpose Symbolic Instruction Code, and was primarily designed as a language that would be easy to learn and use.

One of its better features is that it allows for interaction between student and computer. When the programmer makes an error, the computer reports it immediately. The programmer then has the opportunity to rectify the error and try again. But its loose structure does lead to the trial-and-error approach. Many BASIC programmers set out with only a vague idea of the program they want to develop — usually little more than a few notes on paper, setting out what input the program will require and what output is expected. After all, why bother with doing more when you need only sit down at your keyboard and begin entering program lines? After entering some lines RUN them, and when something is wrong the computer will tell you anyway . . .

It is true that many BASIC programmers derive great joy from this approach — it often leads to great ideas and sometimes truly ingenious program solutions. But, by the same token, it can be a source of great frustration, because the programmer often gives up in despair when a certain programming problem continues to elude solution. Many give up computing altogether. Those who persist may eventually master their computer and its language this way, but it is a painstaking way to learn.

It is quite normal for a program to need improvements in the form of extensions, alterations and other updates. The authors of these trial-and-error programs then face a fresh challenge. The lack of structure and documentation makes them finally admit that they no longer remember how they designed the program. This leads to an at least partial rewrite of the program, thus consuming more time and energy. The task is even more difficult if someone other than the author has to accomplish such an update.

1.5 The characteristics of Structured Programming

The three main characteristics of structured programming are:

- The program as a whole consists of a main program (or 'Mainline') and a number of program blocks, variously described as 'modules', 'sub-progams', or 'subroutines', and sometimes simply 'routines'. Each subroutine is designed as a separate program-within-a-program that performs one task only, and ideally can be tested independently of the program as a whole.
- All routines, be they mainline or subroutine, adhere to the rule of the top-down approach: there is only one entry point for the program, at

the top end, and one exit point, invariably at the bottom end. Hence the designation of 'top-down' approach.

- The program as a whole is abundantly documented with appropriate comment lines, where these may not be in the program itself.
  NOTE: example A has a great number of comment lines. In fact the documentation here borders on redundancy. This, however, has been done for the sake of clarity.

1.6 The benefits of Structured Programming

The three major benefits of structured programming are that it is:

- Easy to write and test — it is not uncommon for a structured program to work almost perfectly at the first try-out; in striking contrast to most trial-and-error programs that generally require lots of correction before delivering the desired results.
- Easy to follow — the author, or other programmers, will have little trouble in tracing the program flow, even years after the program was developed. Again, this contrasts with trial-and-error programs that, in a short time, tend to become unrecognisable even to the authors themselves.
- Easy to update, alter and extend — extensions are a simple matter of adding extra subroutines. Alterations and updates require only a partial rewriting of one or more subroutines and not the program as a whole. The overall structure remains unchanged.

1.7 The planning stages of Example A

Example A furnishes an almost perfect specimen of a program that has been developed entirely in accord with the principles of structured programming. Therefore, let us use it as a guide to developing such a program.

To start with, there is the seed idea. In example A this is the question, 'How to convert hours, minutes and seconds to decimal hours?' (there is nothing trivial about such a question — situations occur where arithmetic must be done with time reckoning. It is a lot easier to work with decimal hours instead of the hours, minutes and seconds format). This leads us to the first stage of the programming job:

1. Define the input and the output formats of the program. Input simply means the data the program needs to work with, to 'process' in order to get a correct result, called the output. It is like a factory in that raw materials are entered (=input) to be processed

and the result is a product ready for sale (=output). By defining the input and the output we determine the goal of our project.

2. The next step is to work out the way to reach that goal, or what procedures have to be developed and extended before a result is achieved? A procedure describes a single task, and that is precisely what a subroutine is expected to perform. Hence, single tasks can be assigned to subroutines.

Now let us consider Example A again. The input may be described as follows: accept the input, which is data in the format of hours, minutes and seconds. After processing, an output is expected in the format of decimal hours.

1.8 Planning the Mainline

The Mainline is the central railway station, as it were, from which other subroutines branch off.

Starting the program is the first procedure, and is in many small programs little more than a formality. The RUN command is typed in, the RETURN key pressed, and the program works or 'runs' as programmers prefer to say.

Entering the input data is the second procedure. We can assign a subroutine to accept the input. You may sometimes encounter the word 'parameter' when input data is meant.

Processing the data is the third procedure. Again, we set a subroutine aside for this task. 'Processing' as a general term may be applied to the entire programming, from beginning to end. But what we mean here is that the data as entered (hours, minutes and seconds) has to be retrieved separately before being made ready for calculation. This leads us to the next step.

Performing the calculations is the fourth procedure. A subroutine will be assigned to convert hours, minutes and seconds into decimal hours.

Displaying the result, or output, becomes the last subroutine.

Summarising the procedures we get the following list:

```
START  (RUN the program)
       Accept data (=Input)
       Process the data
       Perform the calculations
       Display result (=Output)
END    (Terminate the program)
```

Program lines 11Ø to 16Ø of example A, the mainline, illustrate the above arrangement of procedures. In the lines 12Ø, 13Ø, 14Ø and 15Ø the BASIC keyword, GOSUB, points to the subroutine that performs the task defined in the early planning stages.

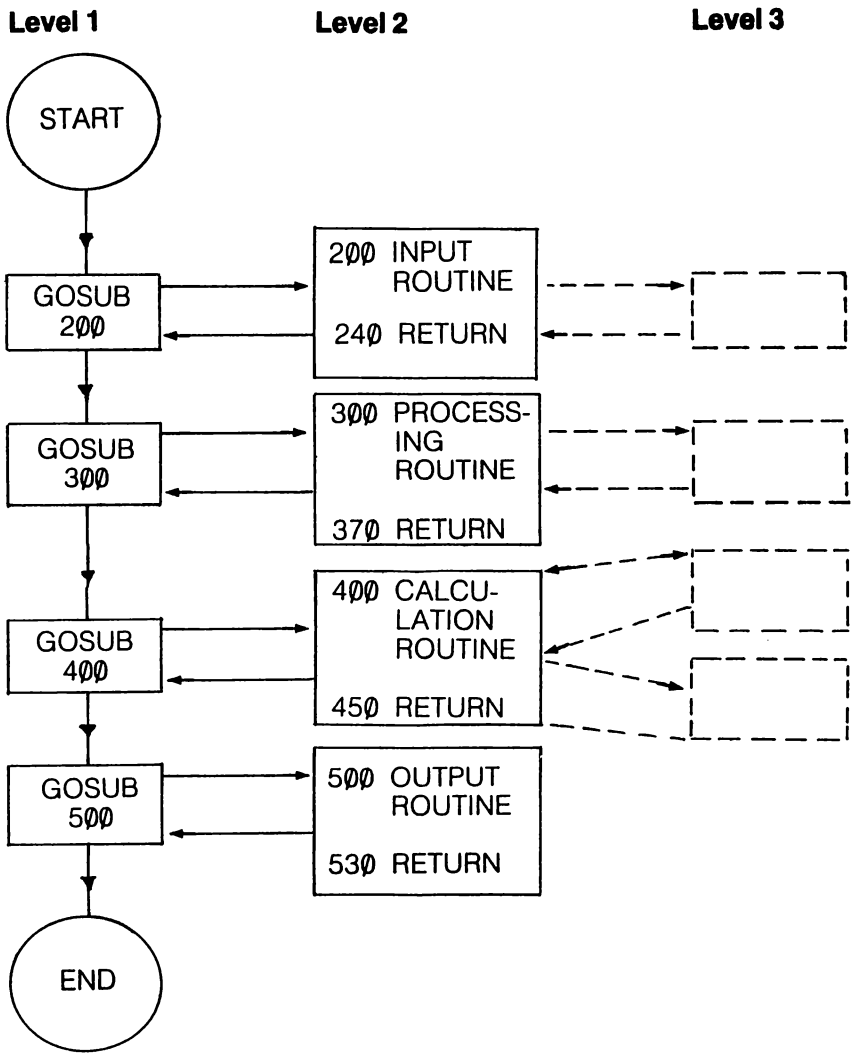
NOTE also that the end of the program is located at the end of the mainline, and not the last line (53Ø). You will find a **return** statement at 53Ø, which merely indicates that the program flow **returns** to the mainline after completing its flow through the subroutine called the 'Output Routine'.

1.9 A diagram to clarify

The next diagram will clarify how the program flow moves through the different subroutines, before coming to a halt at the END statement in line 16Ø. When you examine this diagram take special note of the following:

1. Each subroutine is entered at the top, and is 'exited' at the bottom end by the program flow.
2. Subroutines are categorised in levels. The mainline is the first-level routine, and in this program we see four second-level routines. **But, most importantly**, the system of structured programming allows for extra subroutines. You can add extra subroutines at the second level by extending the mainline. If a second-level routine doesn't already 'activate' a third-level routine, it is easy to create one. Third-level routines may feed fourth-level routines, and so on. In any case, this diagram shows clearly the 'modular' design of structured programs. They are like building blocks; link them up properly and you have a perfect working program.

For these reasons alone, structured programming amply justifies itself.



1.10 What to expect in the next chapters

We are now ready to start our projects, so here is a preview of what you can expect.

The following chapters will provide you with a program concept, its mainline, and an outline of the various subroutines. It is your job to fill out those subroutines with appropriate program lines. It is not hard — you get help all the way. The projects are relatively simple, particularly

projects 1 and 2, which are copiously annotated with explanations, hints and standard program line formats. When you have successfully completed them, it won't be difficult to complete the others.

The projects are manageably simple. A proficient programmer would write some of them in half an hour or less; beginners will need more time, or course. But the projects have been so designed so that each routine should take no more than a few hours to complete.

1.11 Save your programs on tape or diskette!

However, be aware that when you switch off your computer, you lose your program! Hence you are strongly advised to save your programs, no matter what stage of development they might be at, on tape (when you have a Commodore Datasette only) or on diskette.

It is imperative that you make yourself thoroughly familiar with saving procedures! Your user manual will tell you how to do it.

Saving on cassette tape is very easy. Saving on disk is easy as well, but requires a little more preparation. If you find Commodore's explanations of disk handling unclear, then refer to Appendix A of the book 'Discover your Commodore 64'. In any case, when you close a session, save what you have developed to date on disk or tape. The next time it is simply a matter of loading it into the machine again, and continuing the project.

1.12 Errors you can make

Errors happen to everyone. The bulk of them will be typing errors, and when you test-RUN the program they will be reported as 'syntax errors'. You should then try to rectify them straightaway. Throughout the book advice will be given on how to remedy likely syntax errors. By the way, syntax errors are errors in language usage.

There is another kind of error you can make. It is called a 'logic error' and stems from faulty program design. You have a logic error when a program flow does not move in the direction you want it to go. When the result of some arithmetic turns out to be wrong, as another example, it is most likely due to a logic error. The rule is that syntax errors (in BASIC) are reported by the computer, but logic errors may pass by unnoticed. However, they certainly become visible in the output. The projects have been designed to reduce logic errors to a minimum. You will also be warned when they are likely to occur.

1.13 Some practical hints

1. Your best companion in the early stages of your projects is a notepad. Try to work out as much as possible on paper, before you touch the computer. The general rule is to THINK BEFORE YOU PROCEED. A good program is the result of about 70% thinking and 30% working with your computer.
2. Though this book will guide you through all the projects with hints and explanations, the best advice we can give you when you try a program on your own, is to follow the rules of structured programming. Therefore, study thoroughly Example A, and paragraphs 1.5 to 1.1Ø.
3. A good program is readable and sensibly annotated (with REMark lines). Readability is improved by typing spaces between instructions, and using open program lines now and then. The latter is easily achieved. Simply type a line number followed by a colon (:).
4. Number with intervals. As you know program lines must have a number. It is normal procedure in the world of programming to number program lines with definite intervals, generally an interval of ten. This allows the insertion of extra program lines if the need for them arises.
5. The Commodore 64 allows up to eighty characters (a character is a letter, a digit, a graphic symbol, even an open space, etc.) on one program line. However, this can lead to a cluttered, unreadable program (see example B). Keep your program lines brief. Of course, when a program statement is extremely short (i.e. A=1), then you may place more of them on the one line.
6. If you do not have a printer (for listing of programs), then ALWAYS write out your programs for later reference. The projects in this book often refer to each other!
7. Finally, apply the general instructions and suggestions given in the next chapters.

Happy computing!

CHAPTER 2
Birthday and Anniversary Calendar (I)

2.1 The concept

Our first project here is to create a program that will save you the trouble of filling out an ordinary birthday calendar or calendar diary obtainable at any newsagent or bookshop. Designing such a program is a lot more satisfying than merely filling out a calendar. We will be developing a program which, immediately after beginning, will display on the screen the request TYPE IN NR OF MONTH, followed by a question mark and a blinking cursor. You will then type in a month number, e.g. 5, and after pressing the RETURN key the program will display the name of the month followed by important birthdays and anniversaries. When you want to do the same for another month, you can RUN the program again. You may also include an 'another go' option (which however is reserved for chapter 3).

2.2 What a screen will look like

*** BIRTHDAY CALENDAR ***

TYPE IN NR OF MONTH: ? 5

THE MONTH IS MAY
BIRTHDAYS ARE:

Ø2 MARY GREEN

15 JOHN DEE

21 CHRISTOPHER PETERSON

25 WEDDING ANN. MUM AND DAD

2.3 The essential part of the program

As you can see from the screen, the information we want will be in the form

```
Ø2  MARY GREEN
```

This information has to be stored in one way or another before it can be retrieved for display on the screen. There are several ways to do this, but in this project we use the simplest method. Each item is stored in a printline, like

```
562  PRINT "Ø2 MARY GREEN"
```

This printline becomes part of a subroutine and will be displayed as soon as the program flow enters the subroutine. We will examine this more closely in a later paragraph.

2.4 Setting up the Mainline

The program may be made up of four program blocks (or modules or routines. As it is widely used in the computer industry I prefer 'routines', and this name will be maintained throughout the book).

Here is a program flow in words for this mainline:

```
Start of Program
  Initialise
  Accept Information (= Input)
  Display Results    (= Output)
Terminate Program
END
```

The first program block is called the initialisation routine, the second the input routine, the third the output routine, and the last the termination routine. Below is an example of how to set out the mainline routine:

```
1   REM *** MAINLINE ***
5:
1Ø  REM INITIALISE PROGRAM
15  GOSUB1ØØ
2Ø  REM ACCEPT INPUT
25  GOSUB 2ØØ
3Ø  REM DISPLAY OUTPUT
35  GOSUB 3ØØ
4Ø  REM TERMINATE PROGRAM
45  END
```

As you know GOSUB stands for GO to SUBroutine. Thus the lines 15, 25 and 35 direct the program flow to the subroutines which do the actual work. After each task has been completed the program flow returns to the mainline.

2.5 Testing the Mainline

Type in this mainline and RUN it. The computer will respond with UNDEF'D STATEMENT ERROR IN 15, because the subroutine pointed to by line 15 is simply not there. The same applies to lines 25 and 35. So long as this problem remains there will be no way to test properly the functioning of the mainline.

2.6 'Dummy subroutines'

This is where the concept of 'dummy subroutines' comes in. They ensure an unhindered program flow during the various developing stages of a program, and can later be replaced by the real thing. Look at the lines below to see how easy it is:

```
100 PRINT"DUMMY INIT. RTN":RETURN
200 PRINT"DUMMY INPT. RTN":RETURN
300 PRINT"DUMMY OUTP. RTN":RETURN
```

Enter them and your mainline will RUN without a hitch.

NOTE: I strongly recommend that you write a dummy subroutine every time you feel there is a need for one. It can always be replaced and this practice ensures a proper program flow during all the stages of program development.

2.7 How to develop the Initialisation Routine

Now that I have given you a fully developed mainline, the remaining procedures of the program are for you to work out. But, as this is your first major job, I will provide you with many clues.

The purpose of the initialisation routine is just what the name implies — it initialises the program. In large programs this process of initialising comprises a lot of work (as you may find out later) but in a small project like this there is not much to initialise. The routine may clear the screen, for example. It may contain an 'identification section' showing the name of the program and the name of the author. A copyright clause may be included as well. It is good practice to include an identification section at all times.

Consequently the initialisation routine may look like this:

```
100 REM * INITIALISATION ROUTINE *
105 :
110 REM (IDENTIFICATION SECTION)
115 :
120 REM TITLE — SIMPLE BIRTHDAY CALENDAR
125 :
130 REM AUTHOR — (WRITE HERE YOUR NAME)
135 REM DATE WRITTEN — (WRITE HERE DATE)
140 :
145 PRINT "♡": REM CLEAR SCREEN
150 :
155 PRINT" *** BIRTHDAY CALENDAR ***"
160 PRINT:       REM BLANK LINE
165 :
170 RETURN:      REM RETURN TO MAINLINE
175 :
180 :
185 REM----------------------------------
190 :
195 :
```

The lines 145 and 155 contain the essential statements of this routine. In 145 the screen is cleared (see the heart symbol which represents the 'clear screen' command of Commodore BASIC). Line 155 displays the title of the program, followed by a blank line (see 160). In 170 the program flows returns to the mainline.

2.8 Testing the Initialisaton Routine

When you have typed in this routine exactly as it is printed here and RUN it, then the screen will display the title of the program and the names of the remaining dummy subroutines. But of course you may have done something wrong, so an error message may appear on the screen. Here

is a breakdown of all possible errors:

1. You forgot to type a line number
2. You forgot to type the REM in a REM statement
3. You made typing errors, for example, RME instead of REM, PRIN or PRIT instead of PRINT, RETRUN instead of RETURN and so on
4. You typed a semi-colon (;) where a colon (:) was required
5. You forgot quotes in a PRINT statement.

2.9 Bugs and Debugging

In case you haven't yet heard of them, I will now introduce you to a couple of very popular terms in the world of computer programmers. An error in a program is usually referred to as a 'bug' and the art of correcting errors is called 'debugging'. We will use these terms from now on. Keep in mind that bugs can be both syntax errors and logic errors (see chapter 1).

2.10 The Input Routine

This routine is the simplest of all, because the only data required for input is the number of the month to be stored in a variable called NR. BASIC has a specific keyword reserved for matters like this. It is the keyword INPUT. Armed with this knowledge — use of NR as the input variable and the keyword INPUT — you can write this routine yourself. Place it between the lines 2ØØ and 26Ø and do not forget to include the 'screen prompt': TYPE IN NR OF MONTH: .

If you can't find a solution, refer to your C-64 manuals, or, as a last resort, to the Appendix of this book.

```
2ØØ REM * INPUT ROUTINE *
21Ø REM
22Ø REM  WRITE HERE
23Ø REM      YOUR
24Ø REM  INPUT ROUTINE
25Ø REM
26Ø REM
```

2.11 Testing the Input Routine

When you have typed in this input routine, RUN the program for testing.

You may encounter the following bugs (see also paragraph 2.8):

1. You typed a colon where a semi-colon was required
2. You typed IPUT or IPNUT instead of INPUT
3. You forgot the RETURN statement.

2.12 The Output Routine

In this routine you will employ the duo, ON-GOSUB. This combination allows a variable to be placed between ON and GOSUB. The program flow will then go to a subroutine indicated by the number placed in that variable. For example, ON B GOSUB 1ØØØ, 2ØØØ, 3ØØØ will have the effect of directing the program flow to subroutine 1ØØØ when B contains 1, to subroutine 2ØØØ, when B contains 2, and to 3ØØØ when B contains 3.

This particular property of the ON-GOSUB combination becomes very useful in this project. See how it is implemented in the output routine:

```
3ØØ  REM * OUTPUT ROUTINE *
31Ø  :
32Ø  ON NR GOSUB 4ØØ, 44Ø, 48Ø, 52Ø
33Ø  :
34Ø  RETURN: REM RETURN TO MAINLINE
35Ø  :
36Ø  :
37Ø  REM----------------------------
38Ø  :
39Ø  :
```

NOTE: the line 32Ø is incomplete in sofaras the number of subroutines is underrepresented. I ask you to enter the remainder of this line. For clues: see the next paragraphs!

2.13 How it works

Remember that the variable NR has received a number between 1 and 12 (inclusive) in the input routine. The output routine examines in line 32Ø the contents of NR and 'ON' the number in NR it decides to go to one of the twelve subroutines ranging from 4ØØ to 84Ø. Thus when NR contains 1, the program flow goes to subroutine 4ØØ (the first in the row). When NR contains 12, the program flow goes to subroutine 84Ø (the last in the row). When NR contains 5, the program flow goes to the fifth number in the row , and so on.

These twelve subroutines contain the important data of the twelve calendar months and must be considered as third-level subroutines. Consequently, when one of those subroutines has been accessed, the program flow comes back to the output routine and from there returns to the mainline.

2.14 Testing the Output Routine

First of all, enter dummy subroutines relevant to each subroutine number in the row of line 32Ø, for example: 4ØØ PRINT "JANUARY DUMMY": RETURN. Secondly, you may have caused the following bugs:

1. Wrong spelling of GOSUB
2. Wrong spelling of ON (ØN for example — a zero instead of an O)
3. A number in the row which does not correspond to the number of the subroutine it is supposed to match.

See also the list of bugs placed under previous paragraphs.

2.15 The Month Subroutines

These subroutines won't give you many problems, because they consist mostly of printlines. See for example the subroutine for the month of May:

```
56Ø  PRINT:PRINT"THE MONTH IS MAY":PRINT:PRINT"BIRTHDAYS
     ARE:":PRINT
562  PRINT"Ø2 MARY GREEN":PRINT
575  PRINT"15 JOHN DEE":PRINT
581  PRINT"21 CHRISTOPHER PETERSON":PRINT
585  PRINT"25 WEDDING ANN. MUM AND DAD":PRINT
595  RETURN: REM RETURN TO OUTPUT ROUTINE
596  :
597  :
```

I have already explained (see paragraph 2.3) that the simplest way of storing data is to place them in a printline. Now take note of the line numbers in this month routine: they tally with the day number of the brithday in question. For example 562 is the second line after 56Ø and thus represents day 2 of the month of May. Note also that there is more than enough room left for line numbers. Every month has been allocated **forty** line numbers (as you see, the difference between each beginning number of a subroutine is exactly forty), giving you the opportunity to insert birthdays and anniversaries until all days have been filled. Of

course the likelihood of all days of the month being filled is pretty remote, but we have to take that into account. It may also happen that two individuals share the same birthday — in such a case there is no problem in writing two names on the same line.

Finally, have a look a line 56Ø. It has many separate PRINT statements, each taking care of a blank line. The other program lines have stand-alone PRINT statements as well, ensuring that every data item is clearly separated by a blank line from the other, thus guaranteeing readabilitv.

Now that I have given a clear idea of how to set up a month subroutine, you should not have any trouble in setting up the other eleven month subroutines and adapting them to your needs.

2.16 Months without data — what to do

Of course, there may be months without important data. This problem is easily resolved. Simply write the month routine, but instead of inserting a birthday statement you insert a line like "NO DATA AVAILABLE FOR THIS MONTH".

2.17 Terminating the program

Large programs often contain intricate Termination Routines. In a simple project like this, however, a termination routine may be simple as well. When, after finishing subroutine 3ØØ the program flow returns to the mainline (in line 35), the program simply ends in line 45 when it encounters the END statement. Of course this may also be indicated by a printline which will display on the screen the message END OF PROGRAM, or something similar. But your Commodore 64 will display READY and a blinking cursor anyway, thus indicating that the program came to a definite stop.

RUN the program again to assess a new month.

2.18 The overall program pattern

Now that you have finished this first software project, observe the different routines all lined up as in the order below:

1. Mainline
2. Initialisation Routine

3. Input Routine
4. Output Routine
5. Termination Routine
6. January Subroutine
7. February Subroutine
8. March Subroutine
9. April Subroutine
10. May Subroutine
11. June Subroutine
12. July Subroutine
13. August Subroutine
14. September Subroutine
15. October Subroutine
16. November Subroutine
17. December Subroutine

2.19 Logic errors in the final test

Let us assume that you have typed in the entire program and tested it thoroughly for the type of bug known as syntax errors. The program has been thoroughly debugged, but the output is still not right. For example, it may happen sometimes that two lists instead of one are displayed. You have then encountered a logic error or an error in the program's design. Although the computer finds no syntax errors, the program flows to places not specified in the program.

In a case like this — two consecutive months being displayed instead of just one — the logic error is caused by the **omission** of the program line which ends one of the month subroutines. The program flow has missed the mandatory RETURN statement and moved into the next month subroutine and so displayed its contents as well. If this error has not occurred in your program, congratulations! But I advise you to introduce deliberately the logic error we just discussed, to observe the effect. Introducing logic errors deliberately — after the program has been finished and is in good working order — will help you in learning to detect this type of error later on when you are working on more complicated projects.

2.20 Extensions and alternatives

In paragraph 2.1 I hinted at the 'another go' option. Such an option enables you to assess another month without re-RUNning the program. The 'another go' option then becomes part of a new routine to be added to the program. The information placed in the month subroutines may be summarised under the heading 'data base'. Though the way we handle the data in this project is very effective, it would be considered to be rather 'primitive'. Advanced programming techniques require a more sophisticated form of data manipulation. BASIC statements such as DATA, READ and RESTORE provide the keys to appropriate solutions in more complex programs.

Both the 'another go' option and the alternative way of data handling will feature in the next chapter: Birthday and Anniversary Calendar 2.

CHAPTER 3
Birthday Calendar (2)

3.1 An alternative approach

The first project led us to the design of a very simple 'data file', consisting of birthdays and anniversaries and stored (or better: filed) in the form of printlines. This method, simple and effective though it may be in small programs, is not so useful in larger projects. Programs primarily designed with the purpose of handling lots of data (hence 'data processing') never use the printline as a means of storing information. The reason is simple: data in the form of a printline cannot be further manipulated — it can be printed and nothing else. The BASIC computer language has reserved some words that deal specifically with the storage and manipulation of large numbers of data. This chapter will introduce you to the practical application of these words: DATA, READ and RESTORE. You will also apply for the first time the program 'loop' and become acquainted with the IF-THEN combination. The LEFT$ statement is included as well. A special feature is the introduction of the 'flag'.

3.2 The outline of this project

Our new project will use a similar format to the finished project with these differences:

1. The mainline will remain as it is, though at a later stage we may make a small alteration.
2. The initialisation routine may stand as it is.
3. The input routine remains much the same, but will be extended with a series of IF-THEN statements.
4. The output routine will be replaced by a totally different routine.

5. The month subroutines will be replaced by a long list of DATA lines.
6. We will add an 'Another Go' routine.

The overall set-up of this program will be:

a. Mainline
b. Initialisation Routine
c. Input Routine
d. Output Routine
e. Another Go Routine
f. Termination Routine
g. A large block of DATA lines, or a DATA base, or DATA file.

3.3 A few words about DATA

Before we consider the DATA statement I hope you remember the preliminary statement in the introduction: it is assumed that you have made yourself sufficiently familiar with most of the Commodore BASIC keywords. If not, then study your User Guide, the Commodore 64 Programmer's Reference Guide, or the book 'Discover your Commodore 64'.

Back to DATA: the birthday items previously placed in print lines are now to be placed in DATA lines, like this:

```
562 DATA "Ø2 MARY GREEN"
```

But for reasons that will become clearer later on, we have to add the name of the month to this DATA line:

```
562 DATA "MAY Ø2 MARY GREEN"
```

We know that DATA lines allow more than one item, provided each item is separated from the other with a comma. Hence we may write.

```
562 DATA "MAY Ø2 MARY GREEN", "MAY 15 JOHN DEE", "MAY
    21 CHRISTOPHER PETERSON"
```

and so on, as long as the total number of characters does not exceed eighty (including the line number and the DATA statement). Though this filling up of DATA lines is perfectly acceptable insofaras the syntax of BASIC is concerned, programs of this nature are a lot easier to write and modify if a limit is set of one item per DATA line. Thus the format of the DATA file might be:

(line number) DATA "MAY Ø2 MARY GREEN"
(line number) DATA "MAY 15 JOHN DEE"
(line number) DATA "MAY 21 CHRISTOPHER PETERSON"

and so on.

It is worth pointing out here that the information recorded in each line of data may also be regarded as a DATA **string**, to be placed into **string variables** later on.

It is often said of DATA statements that their location within a program is not so crucial, because as 'non-executable' statements they are ignored when encountered by the program flow. However, it is good programming practice to place all DATA statements **at the bottom end** of the program.

3.4 A few words about READ

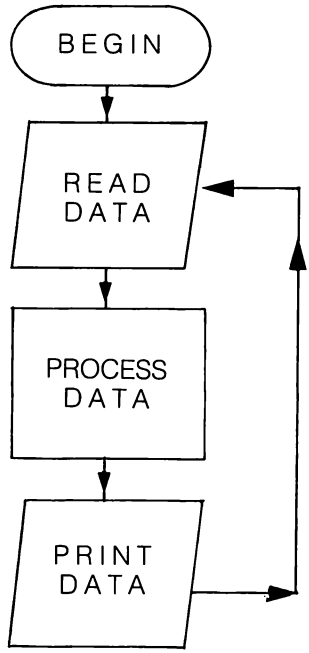
A DATA item can only be retrieved in one way: a DATA item must be **READ.** The READ statement 'looks into' a DATA file and when it finds an item, it will place it into the variable following the READ statement. The next example lines demonstrate this:

```
1ØØØ DATA "JOHN", "PETER", "WILLIAM"
1Ø1Ø READ A$
1Ø2Ø PRINT A$
```

RUN this tiny program, and discover that only one item — the first of the list, John — is displayed. This is because the READ statement reads only one item at a time, and in consecutive order. The computer has a so-called 'data pointer', which at the outset of a program 'points' to the first item of a DATA list. When it is READ, the pointer moves to the next item and waits until this too is READ.

3.5 READing items via a Program Loop

All items in a DATA file can be READ when this statement is placed in a program loop. The fundamentals can be explained in words, but you will grasp the concept sooner if it is presented in visual form. The next diagram shows a 'flowchart' for the READ-DATA operations.



Here you see how, in the first instance, DATA is READ. Next comes the processing of DATA, and after the printout, the program flow returns to the top end of the loop and the entire process starts all over again.

This concept of looping is at the core of our present project.

3.6 Begin the modification of project 1

The largest chunk to be modified is the data base. Replace all printlines that contain birthdays and anniversaries by DATA lines in the format:

DATA "MAY Ø2 MARY GREEN"

IMPORTANT: as we are going to add another subroutine that will start at line 4ØØ, we'd better start our new DATA file at a line-number far removed from 4ØØ. Line number 1ØØØ is a good starting point for your new DATA file. Next, erase all lines stating the names of the months, and erase as well all the lines ending every former month subroutine, i.e. RETURN statements. Do not erase, of course, the RETURN statements at the end of the initialisation and input routines. (Remember how to delete redundant program lines? Simply type the line number and press the RETURN key.)

Finally, erase the entire output routine, except for the entry line (line 3ØØ). For your convenience, delete the RETURN line (34Ø) and insert a new one at line 39Ø. The sector between 3ØØ and 39Ø is now reserved for the new output routine. When you have done all these modifications, save the result on tape or disk for later use.

3.7 Setting up the Output Routine

As we have already discussed how to create an output routine, I shall ask you, the reader, to write it. This entails little more than replacing the appropriate REM statements in the next lines:

```
3ØØ REM *** OUTPUT ROUTINE ***
3Ø5 :
31Ø REM — READ DATA
315 :
32Ø REM — (PLACE HERE YOUR READ STATEMENT)
325 :
35Ø REM — (PLACE HERE YOUR PRINT STATEMENT)
365 :
37Ø REM — (HERE LET YOUR PROGRAM GO BACK TO THE READ
    LINE)
375 :
39Ø RETURN: REM — EXIT TO MAINLINE
```

NOTE: by now you have realised that a DATA item, once READ, must be placed in a variable. Question: What kind of variable when we know that each item of the DATA file is a string?

We will need a name for the variable into which a DATA item is to be placed. I suggest you choose a name that reflects the nature of the data. Commodore BASIC allows variable names up to sixteen characters long, but unfortunately it recognizes only the first two! When you give this variable the name BIRTHDAY$, the program reads BI$. Question: What does the dollar sign here indicate?. Because of this pecularity you cannot use other variable names which begin with the letters BI. This makes it highly advisable to use two-character variables at all times, thus avoiding possible confusion (and program bugs) caused by different variable names, whose first two letters happen to be identical.

3.8 The first test RUN

When you have entered the three requested program lines, give the program a first test RUN. When the computer reports syntax errors these may be due to:

- wrong spelling of READ
- wrong type of READ variable (remember, the program must read strings!)
- wrong spelling of PRINT and GOTO

For other bugs, see the error listings elsewhere.

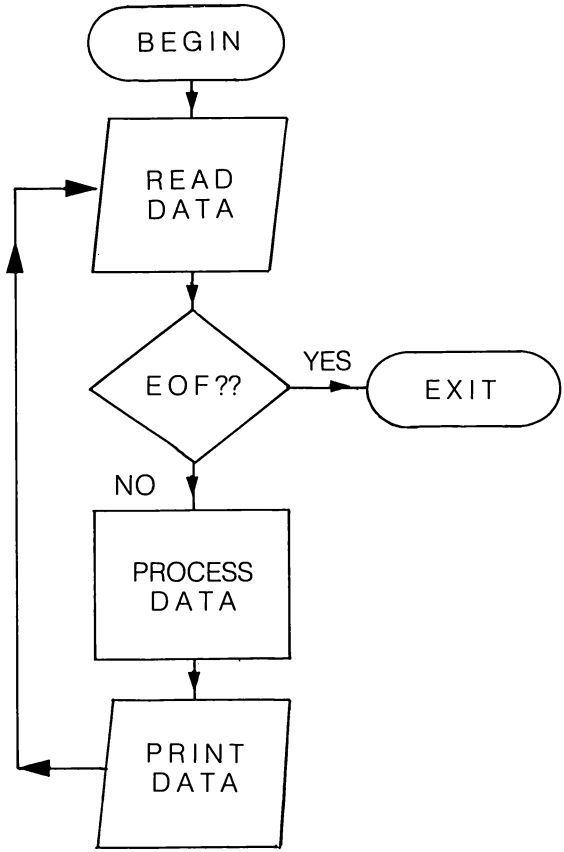
After debugging and running this part of the program as is, you'll be in for a suprise. It will list **all** DATA items (birthdays for the entire year) and then stop with the error message:

OUT OF DATA ERROR IN 32Ø

This is because we have created a logic error, which may be classified as a syntax error as well (the computer reports it). The logic error is caused by the output routine in its present form — it is an **endless loop**, in fact. It would go on running round in circles forever if there were an infinite supply of DATA items. There is not, and when the program has run out of data the computer displays an error message and stops further execution of the program.

3.9 The END OF FILE DATA item

How to resolve this problem? Very easily. The usual solution in a case like this is to create a **last** DATA item, which has one purpose only: to indicate the END OF FILE (normally abbreviated EOF). The next step is to include a so-called 'conditional test' in the READ DATA loop, which during the program RUN, continuously tests whether the EOF item is READ or not. IF this item is READ, THEN the program loop is broken — the program flow branches out. The following flowchart will demonstrate this visually:



Note the diamond shaped symbol in the centre of the flowchart. It represents the conditional test: "is the end of file (EOF) reached?" IF the condition is "true" (= yes) THEN the program flow will branch out of the loop — here to the exit of the routine. If the condition is "false" (not true = no) then the condition is not met and the program flow remains in the loop. We will now implement this new knowledge. **First of all, create a DATA line containing the EOF item.**

It is advisable to give this last DATA item a very high line number, thus ensuring that you may insert as many real DATA items as you wish between the last real item and the EOF one. A conspicuous line number like 9999 will do the job perfectly.

Now consider the next listing. Replace the appropriate REM statement with the correct program line.

```
300 REM *** OUTPUT ROUTINE ***
305 :
310 REM — READ DATA
315 :
320 REM — (PLACE HERE YOUR READ STATEMENT)
325 :
330 REM — (PLACE HERE YOUR EOF TEST AND MAKE SURE TO
    BRANCH TO THE EXIT POINT OF THIS ROUTINE IF THE TEST IS
    TRUE)
345 :
350 REM — (PLACE HERE YOUR PRINT STATEMENT)
365 :
370 REM — (LET HERE YOUR PROGRAM GO BACK TO THE READ
    LINE)
375 :
390 RETURN: REM — EXIT TO THE MAINLINE
```

3.10 The second test RUN

RUN the program to test this stage of project 2. If a syntax error is reported, it may have been caused by:

- wrong spelling of the words IF and THEN
- forgetting to place EOF between quotes

A possible logic error may be that you have directed the branched-out flow to the wrong exit line (remember: any subroutine is supposed to exit at a RETURN statement).

When you have entered the conditional statement correctly, the program will RUN without the OUT OF DATA ERROR message. But does it now give the desired result? The answer is no, because the program delivers all available data, scrolling over the screen when there is much of it. We are looking for an output that delivers only the birthdays and anniversaries of a month specified at the input, just as we did in project 1.

3.11 The concept of the Record Key

So the next problem we have to solve is how to get only the desired data on the screen. A very elegant solution is offered by the so-called 'Record Key'. We have spoken about files and that files may be READ. We shall now designate a large collection of data a 'file', and each item of data within the file a 'record'. In other words, a file contains records.

To elaborate: when you have placed a birthday in a DATA file, you have recorded that birthday and it therefore becomes a record.

Now, to **identify** a record, we must have a 'key'. That notion is easy to understand. For example, your savings account at the bank has as its record key your account number.

3.12 The name of the month as Record Key

If you have been following our argument closely you will have guessed that I asked you to include the name of the month in every DATA item, because it is the month's name that will function as the key to the output. How is this to be done?

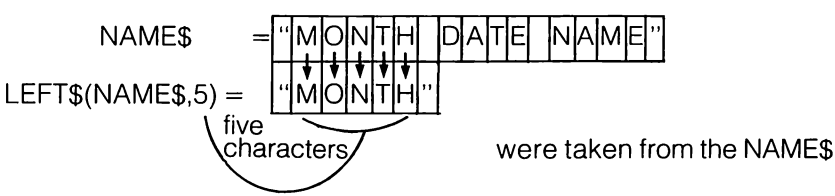
The principle is: READ the DATA item and try to 'match' the record key — here the name of the month — with the name of the month you just placed in the program via the input routine. If the two names match, only then will the record — here a birthday or anniversary — be displayed.

Now to demonstrate.

3.13 We manipulate strings

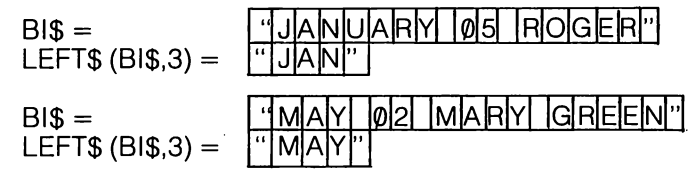
A string like "MAY Ø2 MARY GREEN" contains many items in one record: the name of the month, the day number, a christian name and a surname. On top of that there are blanks between those items, and blanks are also regarded by the computer as data items. Easy as it may be to manipulate simple variables containing only numbers, string variables require closer attention. BASIC has reserved special keywords for the manipulation of strings. Your User Guide and the book, 'Discover your Commodore 64', will give you lots of information about string handling but here we limit ourselves to the keyword LEFT$.

(By the way, the dollar sign is usually pronounced 'string' in the context of the BASIC computer language. Thus the statement, LEFT$, becomes "LEFT STRING".) When we use LEFT$, we will be able to pick out the section of the record we need. Consider the diagram below:



Here you see how the LEFT$ statement picks out the five left-m characters from the string and places them into another variable. digit 5 in KY$ = LEFT$ (BI$,5) indicates that five characters, beginn from the left, are read and placed into the string variable KY$ (reflects 'key'). It is the contents of the KY$ variable which are to matched with the **Input Variable** (but more about that later).

You may have already observed a problem here: whereas the LEF statement works only with a pre-specified length definition, the names the months are not of equal length. While it is possible to write a progra including a LEFT$ statement that takes into account the varying lengt of the month, such a program would be unnecessarily complex. T solution we shall implement here is very simple. Consider this:



As you see the first three letters of each month will serve us adequate enough as a record key.

Hence we may extend the output routine with this line:

```
350  KY$ = LEFT$ (BI$,3)
```

The variable KY$ must now be tested on its content, and depending c this the program will print a birthday, yes or no. But testing the conte of a variable means comparing it with something else. And this is wher the **Input Variable** comes in.

3.14 More about the Input Variable

As we know from project 1, it is the input that determines the output. For example, when you were prompted to type in the month number 5, say all the birthdays of the fifth month — May — appeared on the screen. The number 5 was stored in a variable called NR, and its contents were later used to determine the course of the program flow.

We will do something similar in this project, 2. We can still use NR, but only as the **initial** INPUT variable because as it contains a number it cannot be matched against the record key variable KY$. Numerics do not match with alphabetics and a syntax error would be the result. This, however, is far less problematic then it looks.

3.15 Extending the Input Routine

The IF-THEN test will resolve this problem. We introduce a second input variable but now one set up to contain a string, and the contents of this string variable are compared with the record key. We will call this second input variable MO$ (MO reflects MOnth).

Now extend your input routine with twelve of the following lines:

```
230 IF NR = 1 THEN MO$ = "JAN"
235 IF NR = 2 THEN MO$ = "FEB"
```

and so on

Close this extended input routine with a RETURN statement at line 290.

```
200 REM *** INPUT ROUTINE ***
205 PRINT "♡":PRINT: REM — CLR SCR & BLANK LINE
210 INPUT "TYPE IN NR OF MONTH: "; NR
215 :
220 REM — TEST OF MONTH NUMBERS
225 :
230 REM — (PLACE FROM HERE ON YOUR MONTH NUMBER
    TESTS)
290 RETURN: REM — RETURN TO MAINLINE
295 :
```

3.16 Now the test of the Record Key

Now that we have a matching type of variable for the input we can set up a test for the record key (enter this line):

```
355 IF KY$ = MO$ THEN PRINT BI$
```

REMEMBER: MO$ is the (second) Input Variable
KY$ is the Record Key
BI$ is the entire Record — the birthday or anniversary, stored as a DATA line.

In this line the record key and the input variable are compared. Only when they match, will the birthday be displayed!

3.17 The third test-RUN

Now RUN the program to test whether you have entered these new program lines correctly. If bugs are reported they may be due to:

- wrong spelling of keywords
- forgetting to place string variables between quotes
- typing of a zero (Ø) where an O is required
- forgetting to add the dollar sign to a string variable
- using wrong string variables caused by typing errors

For other bugs refer to the error listings elsewhere. When there are no syntax errors, the program will display on request the birth data for the month as indicated by the number input.

3.18 A shorter Record Key test

Although the present test of the record key in lines 35Ø and 355 is perfectly satisfactory, it is not the only way of achieving the same result. The separate record key variable KY$ may be dispensed with, for example, and the test applied directly:

```
35Ø IF LEFT$ (BI$,3) = MO$ THEN PRINT BI$
```

It is up to you whether or not to opt for the more elegant solution. But if you decide on the latter and enter the revised line 35Ø, you must then consider what other lines have to be deleted.

3.19 Adding the Another Go routine

The program in its present state must be re-RUN every time you wish to input a fresh birth date. As I pointed out in chapter 2, this is not a very sophisticated way of designing a program that is supposed to deliver data from a relatively large data base. Let us then turn to a routine that displays a message on the screen after the data of a particular month has been delivered. The message is: "Another Go? Type Y or N and press RETURN". Pressing the Y or N key and then the RETURN key allows you to either return to the input routine (if "Y") or end the program.

3.20 We must extend the Mainline

As we are about to add another major routine, we must extend the mainline. Structured programming, you will recall, requires that all major routines, or 'modules', have to be linked up via a mainline routine. This is the strength of structured programming — a new routine can easily be added to the program, thanks to its modular structure. Now, look again at the mainline as depicted in diagram 2 (see previous chapter).

The mainline ends at lines 40 and 45. Delete these lines and replace them with:

```
40 REM — ANOTHER GO OPTION
45 GOSUB 400
50 REM — TERMINATE PROGRAM
55 END
```

3.21 Setting up the Another Go routine

You will deduce from the foregoing paragraph that the another go routine begins at line 400. This small routine will fit easily within the range 400-500, so reserve that space for it.

Now consider again the screen prompt:

ANOTHER GO? TYPE Y OR N AND PRESS RETURN

Some of the things that should be passing through your mind at this stage are:

1. The "TYPE Y AND PRESS RETURN" prompt clearly suggests an INPUT situation.
2. The fact that the input is either the letter Y or N, indicates that this input must be stored in a string variable, hence of the type "A$".
3. The Y or N decision suggests conditional testing in the form of IF-THEN.
4. This routine is in fact a subroutine, addressed from the mainline. Hence it must end with a RETURN statement.
5. The rules of structured programming require that every major subroutine is a program in itself, with one entry point and one exit point.

Point 5 could put us in a rather 'tricky' situation, as we will discover later.

3.22 Important considerations

Let us look more closely at the preceding five points in the light of the principles of structured programming. The best way to tackle the problem is to sketch out the program flow in words:

```
On  Screen Prompt enter Y or N
      IF Y THEN RUN
      IF N THEN END
If    Other than N or Y go back to
      Input at Screen prompt
```

NOTE: the last sentence of this 'verbal flowchart' takes care of typing errors. If the user presses a key other than Y or N, the program flow 'drops through' and moves into an unwanted direction. The last sentence obviates such a possibility.

At INPUT, the Y or N will be stored in a string variable and its contents examined via a conditional test (IF-THEN). When the test on Y proves to be true, the program flow encounters RUN. This command resets the computer entirely, including the data-pointer we talked about in paragraph 3.3, and 'clears' all variables, which simply means that they are all set to zero; the numerical variable NR then contains Ø and a string variable like MO$ is likewise void of content. This being done, the program RUNs again. On the other hand, when the test on N proves to be true, the program flow encounters the statement END, and comes to an abrupt halt.

Effective as this may be, it is far from elegant. Worse, it does not comply with the rules of the top-down approach in structured programming. Reason: not only are there two points of exit in the above routine, but it also halts in the middle of a subroutine instead of in the termination routine at the end of the mainline. The RETURN statement in the lower part of the another go routine is then rendered useless, as is the GOSUB 4ØØ in the mainline.

Will this work?:

```
On  Screen Prompt enter Y or N
    IF Y THEN GO TO EXIT
    IF N THEN GO TO EXIT
If  Other than Y or N then go back
    to Input at Screen Prompt
Exit
```

The answer is a definite 'no'! In both cases — Y and N — the program flow will move to the exit point (i.e. the RETURN statement at the end of the routine), come back in the mainline and flow straight into the termination routine. This is despite the fact that the another go routine now has only one entry point and one exit point. It is clear that we must insert something between the conditional test and the exit point that produces the desired program flow. And so we are led to the concept of 'Flags'.

3.23 'Flags' as the traffic signals in programming

The 'Flag' concept may be compared with traffic signs on the road. Is the traffic light red? If yes, stop. If green, continue. If amber, prepare to

stop if there is enough time for it — if not, continue. In computer programming we do something similar with flags: a flag can be 'set', and a flag can be 'cleared'. Or, in more everyday language, a flag is up or a flag is down.

The idea is that once a flag is set in a program, it can be tested anywhere in that program. For example, you can set a flag at the beginning of a program and at a totally different place the program flow will encounter the test, "Is the flag set? If yes then go to . . .; If no then go to . . .". Later on the flag can be cleared, and when the program flow again encounters a flag status, the decision may be different. BASIC even allows you to have a program line like this:

```
IF FLAG = 1 THEN FLAG = Ø
```

In such a case a flag clears itself.

In practice a flag is nothing but a variable, whose content is to be tested.

3.24 A Flag in the Another Go routine

The flag will enable us to set the course of the program flow at the outset of decision making in the another go routine.

Consider the following verbal flowchart:

```
On  Screen Prompt enter Y or N
    IF Y then set Flag and Go to Exit
    If N then clear Flag and Go to Exit
If  Other than Y or N go back to Screen Prompt
    Exit
```

When we implement this solution we will also create a program line to test the flag. This test **must** be placed in the mainline! Why? Simply because if you create a separate subroutine for it, you must create an extra flag as well! If the test is true, a simple command will direct the program flow to go to that part of the mainline which points to the input routine.

If the test is false (= not true), the program flow will move on to the termination routine.

Let's see how we may write program lines that do the job as described in our verbal flowchart:

```
IF FG$ = "Y" THEN F=1: GOTO (follows line number of RETURN)
IF FG$ = "N" THEN F=Ø: GOTO (follows line number of RETURN)
```

And in the mainline:

IF F = 1 THEN (follows the line number of the GOSUB pointing to the input routine)

NOTE that it is not at all necessary to enter the test IF F = Ø THEN . . . Why not? (If you can't find the answer, consult the Appendix.)

3.25 Writing the Another Go routine

You should now be ready to write your own another go routine. Replace the appropriate REM statements of the next diagram with the appropriate program lines:

```
4ØØ REM *** ANOTHER GO ROUTINE ***
4Ø5 :
41Ø PRINT: REM – BLANK LINE (WHY?)
415 :
42Ø REM — (PLACE HERE YOUR INPUT STATEMENT INCLUDING
    THE SCREEN PROMPT)
425 REM — (PLACE HERE YOUR FIRST CONDITIONAL TEST (ON Y) )
43Ø REM – (PLACE HERE YOUR SECOND CONDITIONAL TEST
    (ON N) )
435 REM — (ENABLE THE PROGRAM FLOW TO GO BACK TO THE
    SCREEN PROMPT)
44Ø :
45Ø RETURN: REM – EXIT TO MAINLINE
46Ø :
```

NOTE: you may give the input string variable of this another go routine any name but, as I have indicated above, it is best to seek out a letter combination which suggests the type of input. You may give the flag any name as well. With programs using many flags, however, you will have to set out the name list carefully, for example, F1, F2, F3, F4, F5 and so on. In this instance we are using only one flag and hence may call it simply, F.

3.26 An altered Mainline

For your convenience I have already changed the mainline. See the next diagram and transfer the changes to your own program.

```
  5 REM *** MAINLINE ***
  7 :
 10 REM — INITIALISE PROGRAM
 15 GOSUB 100
 20 REM — ACCEPT INPUT
 25 GOSUB 200
 30 REM — DISPLAY OUTPUT
 35 GOSUB 300
 40 REM — ANOTHER GO ROUTINE
 45 GOSUB 400
 50 REM — TEST OF FLAG
 55 IF F = 1 THEN 25
 60 REM — TERMINATION ROUTINE
 65 END
 70 :
```

3.27 The fourth test RUN

Having entered the new routines we can go for another test-Run. For syntax errors, consult the error listings in this and the previous chapter. Examine not only the new another go routine, but also the new entries into the mainline.

You will probably find there's at least one reported error that defies all your attempts at solution. When you run the another go option, by pressing Y and RETURN, the computer will very likely respond with the message: OUT OF DATA ERROR IN 320. This leads us to the RESTORE statement.

3.28 Without RESTORE it won't work

I mentioned the data-pointer in an earlier paragraph. It moves down the list of DATA items, every time the DATA list is READ, until it reaches the last item. If you want to READ those items again, the data-pointer must be RESTOREd to the top of the DATA list. The OUT OF DATA ERROR is simply explained by the fact that after one RUN the data-pointer had arrived at the last item (="EOF") on the DATA list and needed to be RESTOREd to the top of the list before commencing to READ the DATA list again.

So a very important BASIC keyword has to be added to our another go routine. It is RESTORE, but where to place it?

Remember the line under paragraph 3.23:

IF FG$ = "Y" THEN F = 1: GOTO (etc)

This line can also be written this way:

IF FG$ = "Y" THEN RESTORE: F = 1: GOTO (etc)

Now alter your another go routine accordingly, and the program will run perfectly.

3.29 An alternative with Get

We now have another go routine which works quite well, but an even more elegant solution is available with the GET command. In the book 'Discover your Commodore 64', the GET command and its applications are clearly set out. Study the possibilities of this alternative carefully and try to apply it to this project.

3.30 Conclusion

Chapters 2 and 3 have supplied you with the tools you will need to construct a workable program. Major principles of programming — looping, conditional tests, a data base — have been covered extensively.

The projects in the following chapters will explore and develop the skills required so far, with less stress on explanations.

CHAPTER 4
Telephone Directory

4.1 The concept

This project again explores the setting up of a DATA file — here a telephone directory — and how to retrieve records from it. Most of the principles outlined in chapters 2 and 3 can be re-applied. At the same time I introduce you to the BASIC statement "LEN", while at the same time we will have a closer look at the GET command. The purpose of this project is a program that, after entering the name of a person, displays his or her name and the telephone number.

4.2 What a screen may look like

After initialising the program it not only displays the title, but also prompts the user to enter a surname and an initial. Having done that and pressed the RETURN key, it displays the wanted information. And of course there is the usual "another go" prompt. See diagram 17.

```
*** TELEPHONE DIRECTORY ***

SURNAME          : ? JOHNSON

FIRST INITIAL    : ? P


JOHNSON/P        (Ø1) 58 6123

ANOTHER GO ? Y OR N
```

4.3 A telephone number entry

A DATA line may look similar to a real line from a telephone directory:

```
DATA "DAVIS  R.      345 6789Ø"
```

Of course there is the omission of the address, but nothing prevents you from adding that to this data line as well. NOTE, however, that a data line may not become too long. If it does, it may not fit within the width of the screen and you get an overflow to the next line. NOTE also that Commodore BASIC allows you to enter strings **without** quotes into a DATA line. This, however, is a dangerous practice when you leave in commas such as in the DATA line above. (Why? Find out for yourself.)

Obviously this telephone directory forms the core of the project, hence the routine which READs and PRINTs them will be the more important one. But let us first consider the general set-up.

4.4 Setting up the Mainline

This is the mainline set out in a verbal flowchart:

```
Start  Program
       Accept Input
       Read and Print data
For    Another Go return to Input
       Else Terminate program
```

Translated into BASIC program lines, it reads:

```
10 REM *** MAINLINE ***
20 GOSUB 100:REM — TO INIT. RTN
30 GOSUB 200:REM — TO INPUT RTN
40 GOSUB 300:REM — TO READ & PRINT RTN
50 GOSUB 400:REM — TO ANOTHER GO RTN
70 IF FL = 1    THEN 30
80 END          :REM — TERMINATE PROGRAM
90 :
```

Now enter this mainline routine.

4.5 Test-RUNning the Mainline

Make it a habit to test-RUN immediately every routine you have just entered. Hence you can now test-RUN your mainline. However, you will recall that a mainline addressing to subroutines which are not yet developed, won't work. Therefore do not forget to enter as well the 'dummy subroutines' we discussed in chapter 1.

Do not worry about the flag test in line 70. No flag has been set and the program flow simply drops through and stops at the END statement.

4.6 Setting up the Initialisation Routine

The initialisation routine starts at line 1ØØ, and may be similar to the ones you created in the previous projects. But if you want to alter it, do so by all means.

```
1ØØ REM * INITIALISATION ROUTINE *
1Ø5 :
11Ø REM (PLACE HERE YOUR INIT. RTN)
115 :
19Ø RETURN :REM — RETURN TO MAINLINE
```

4.7 Input: entering surname and first initial

The input routine starts at line 2ØØ and it will accept a surname and the first initial. Why only the first initial? Of course you may design an input routine that accepts more than one initial, but there is a fair chance that you know most of your friends and acquaintances by their first initial anyway, hence there is no need to complicate the program on this point.

Both surname and initial form together the key with which the records from your DATA file are to be matched. There are two ways to enter this surname and the first initial.

The first method requires input in one go: "Surname I".

The second method requires a separate entry of the surname and the first initial. At first sight the first method seems to be the most efficient. However, efficient as it may look, it may nonetheless give rise to some minor problems.

There is a maxim in the world of computer programming stating that it is best to keep the input as easy as possible. Well, the first method demands that the sequence of data is entered absolutely correctly. If, for example, you typed two spaces instead of one between surname and the initial, the result may be that the program won't recognise the name as one stored in the data base!

It is therefore advisable to set up an input format that accepts the surname and the first initial separately. Hence the double screen prompt as depicted on page 43. You then only have to type the surname, press RETURN, type the initial without any extras such as spaces or periods, and press RETURN again.

Both the surname and the initial are 'alphabetics', hence must be stored in string variables. SN$ (for SurName) may be a good one for the surname and FI$ (for First Initial) may be a proper variable name as well.

The next step is to 'concatenate' SN$ and FI$. This means that the two are linked together to form a new string, that can be named NM$. This is simply done by adding up: NM$ = SN$ + FI$.

However, you won't like the result. A name like P. Johnson will be stored in NM$ as JOHNSONP. It is better to create a name string which has a space between the surname and the initial, or something else, such as a slash (/). The BASIC statement then becomes NM$ = SN$ + "/" + FI$.

The name P. Johnson will be stored as JOHNSON/P.

Now that you are armed with these suggestions, you are ready to enter your input routine. See the listing below:

```
200 REM * INPUT ROUTINE *
205 :
210 REM (HERE FIRST INPUT + SCREEN PROMPT)
220 REM (HERE SECOND INPUT + SCREEN PROMPT)
230 REM (CONCATENATE HERE THE ABOVE STRING VARIABLES
    INCLUDE A CONNECTION SYMBOL)
240 PRINT   : REM — PRINT A BLANK LINE
250 RETURN: REM — RETURN TO MAINLINE
```

4.8 The BASIC statement LEN

In this program the surname, combined with the first initial and a connection symbol (such as the slash) functions as the record key. However, this record key varies in length as obviously all surnames are different. The solution to the problem of how to assess a record key that continuously varies in length can be resolved by the BASIC statement "LEN". LEN stands for LENgth and it is used in the format X = LEN(Y$). It simply counts the number of characters contained in a string (including spaces). For example, the expression X = LEN("LONDON") stores the digit 6 in variable X because the name "LONDON" contains six characters.

So if you want to establish the length of the combination surname-connection symbol-first initial (stored in NM$), simply write something like this: LG = LEN(NM$).

4.9 The Read and Print routine

The read and print routine starts at line 3ØØ and may be written in similar fashion to the output routine of project 2. But add the following extensions and alterations:

1. Establish the length of the combination surname-connection symbol-first initial and store the result in an appropriate variable (see paragraph 4.8)
2. Substitute the **numeric** in the LEFT$ statement with the variable containing the length of the record key (see point 1).
3. Substitute MO$ with NM$.
4. Give appropriate names to the string variables containing the record (REMEMBER: the record is the data from the DATA line, comprising a surname, its first initial and the telephone number).

Now write this routine and use the next listing as your guide.

```
3ØØ REM * READ & PRINT RTN *
3Ø5 :
31Ø REM (ESTABLISH LENGTH OF NM$)
32Ø REM (HERE YOU READ STATEMENT)
33Ø REM (ESTABLISH LEFT STRING OF DATA RECORD HERE)
34Ø REM (TEST LEFT STRING DATA WITH NAME STRING AND PRINT
    IF MATCH)
35Ø REM (PLACE YOUR END OF FILE TEST)
36Ø REM (REMAIN IN THE LOOP — DIRECT TO LINE 32Ø)
37Ø :
38Ø :
39Ø RETURN : REM — RETURN TO MAINLINE
```

4.10 No test-RUN without a DATA-list!

The read and print routine can only be RUN and tested successfully when it can access a DATA file. Therefore you must set up at least the beginning of such a file. Start the DATA list at line 1ØØØ, enter one complete record per DATA line, and again use as the last item the "EOF" name, in line 9999. Enter a telephone record as outlined in paragraph 4.3

4.11 A test-RUN

I assume that you have already tested your input routine and found it to be without errors. If any bugs were reported, however, you might have

referred to the error lists elsewhere and consequently debugged it successfully.

The same remarks apply to the read and print routine. If you have got project 2 working, there is not much that could have gone wrong this time. In most programs of this size and slight complexity, most bugs are only due to typing errors. A source of problems may be the choice of variable names.

Always keep track of all the variable names you create. Avoid double use! Otherwise you will be in for 'logic errors' that often take hours and even days to be rectified! Avoid also variable names that are almost similar, for example NM$, MN$, MM$ and NN$. They are often confused because at a glance the letters look the same.

4.12 "GET" in the Another Go routine

Have you followed up my suggestion at the end of Chapter 3 to apply the GET command in the another go routine? If you got it working, congratulations! If you weren't successful or declined to try it out, then this paragraph will explain it all.

The function of GET is to 'get' a character from the keyboard of the computer.

It is usually done in this way:

```
3000  GET X$
3010  IF X$ = " " THEN 3000
3020  IF X$ = "A" THEN 4000
```

At line 3000 the computer expects a character entered from the keyboard to be stored in X$. As long as that does not happen, the contents of X$ are void, which is found out in line 3010: If X$ is empty then go back to line 3000. What happens is that as long as the keyboard is not touched, the program is locked in an endless loop in lines 3000 and 3010. It can only branch out of this loop as soon as the keyboard is touched.

This makes the GET command extremely useful:

1. When used in lines 3000 and 3010 it does not stop the program flow, but to the user it seems as if the program has come to a halt anyway. The program seems to pause. Actually, it **waits** for a **manual** instruction, i.e. by a key stroke.

2. When no specific command comes after line 3Ø1Ø, any key stroke will break the vicious circle and the program can go on. This is particularly useful in programs that display text on the screen. A text block ends with a GET sequence of statements as in lines 3ØØØ and 3Ø1Ø, and the text can remain on the screen for as long as the reader want it. Pressing any key will cause the program to display another block of text, and so on.
3. The key entry via GET X$ can also be tested via an IF-THEN test — see line 3Ø2Ø. This is particularly useful in the setting up of a '**job menu**', which leads us to the use of GET in the another go routine. You can write this as follows.

(verbal flowchart)
Display prompt: Another Go, Y or N
1. Get a character from keyboard
   If character equals Y then set flag and go to exit
   If character equals N then clear flag and go to exit
   If not Y or N then return to 1.

Exit.

When you work this out in BASIC program lines, you will see that it is very similar to the another go routine of project 2 (chapter 3). However, the input line with the screen prompt must be replaced with a separate screen prompt line and two lines similar to 3ØØØ and 3Ø1Ø in this paragraph. So it is now up to you to fill out the listing below:

```
4ØØ  REM * ANOTHER GO RTN *
4Ø5  :
41Ø  REM (PRINT HERE YOUR SCREEN PROMPT)
42Ø  REM (PLACE HERE YOUR GET STATEMENT
43Ø  REM (TEST CONTENTS OF STR. VAR. IF EMPTY DIRECT TO 42Ø)
44Ø  REM (TEST STR.VAR.FOR Y)  (and don't forget the flag!)
45Ø  REM (TEST STR.VAR.FOR N)  (and don't forget the flag!)
46Ø  REM (IF NEITHER Y NOR N BACK TO 42Ø)
47Ø  RETURN : REM — RETURN TO MAINLINE
48Ø :
49Ø :
```

4.13 The last test-RUN

When you have successfully completed the another go routine in chapter 3, there is not much that can go wrong here. The usual syntax errors may occur because of incorrect typing. Have you checked that string variables are indeed string variables and not by any chance a numeric variable because you forgot to place a dollar sign after it?

What happens, by the way, when you do not direct the program flow back to the GET A$ line but, for example, to the screen prompt line? Try this out!

4.14 Conclusion

After you have got the program running allright, the only thing that remains is to fill out the rest of the DATA lines. Of course it is easy to update the program later on, just by adding new names and telephone numbers, deleting obsolete ones and then resaving the program, either on tape or disk.

You will have appreciated the simplicity of this project and maybe did not find it much of a challenge. Don't worry, the next projects are harder nuts to crack!

CHAPTER 5
Mortgage Repayment

5.1 The concept

Computer programs that calculate mortgage repayments enjoy great popularity, because not only do they enable home owners or home buyers, to assess exactly how much they have to repay the bank every month, but obviously they are invaluable in the planning of the household budget. After all, the monthly mortgage repayment usually cuts deeply into the breadwinner's take home pay.

Therefore, a mortgage repayment program requires the following data as input:

1. the borrowed amount.
2. the duration of the mortgage.
3. interest rate per year (or per 'annum' as one prefers to say).

The output will be the monthly repayment. This can be extended to yearly repayment, and if you want to find out how much the bank earns from you, the program will also calculate how much in total you will pay borrowed amount, and the duration of the mortgage. By the way, the mortgage formula to be worked out in this project pertains to the 'flat interest rate' option of mortgage repayment!

5.2 What a screen may look like

```
BORROWED AMOUNT (IN $$)      : 4ØØØØØ
DURATION OF MORTGAGE (YRS) : 3Ø
ANNUAL INTEREST RATE (%)     : 12
```

```
MONTHLY REPAYMENTS          : 413.81
APPRX. WEEKLY REPAYMENTS    : 95.22
TOTAL OVER 3Ø YEARS         : 148972.39


(I)NTEREST RATE
(B)ORROWED AMOUNT
(D)URATION
(E)XIT

PRESS APPROPRIATE LETTER FOR VARIATIONS
```

The upper part of the screen shows the input, the centre shows the result of the calculations, and the lowest part gives you a choice of options, the so-called 'Menu'. NOTE as an extra option the word 'exit'. When you press the E the program will terminate. So in effect this options menu is nothing else but an extended another go prompt. NOTE as well that the absence of an input question mark shows the employment of the GET command as introduced in project 3.

5.3 The overall set-up

This project won't introduce you to keywords other than the ones we have already made work in the previous projects. However, the translation of a mathematical formula into a BASIC statement that the computer not only understands but also works out correctly, will be a novelty for you. Also new to you will be a "DEF FN" function, to be placed in the initialisation routine.

The program may consist of the following routines:

1. Initialisation
2. Input Routine
3. Calculation Routine
4. Output Routine
5. Options Routine

As the options routine apparently has three important variations to offer, we may well include three extra routines:

6. Alternative Interest Rate
7. Alternative Borrowed Amount
8. Alternative Duration

This leads us directly to the mainline of this project, already prepared for you in BASIC:

```
10 REM *** MAINLINE ***
20 GOSUB 100        :REM — TO INIT. ROUTINE
30 GOSUB 200        :REM — TO INPUT ROUTINE
40 GOSUB 300        :REM — TO CALC. ROUTINE
50 GOSUB 400        :REM — TO DISPL. ROUTINE
60 GOSUB 500        :REM — TO OPTION ROUTINE
70 IF FL=1 THEN 40  :REM — TEST OF FLAG
80 END              :REM — THE PROGRAM ENDS
                     HERE
```

Enter this mainline routine, and do not forget to test-RUN it (enter dummy subroutines!).

5.4 Rounding and DEF FN

We are about to enter the initialisation routine but, before we do that, let us have a closer look at the screen lay-out under paragraph 5.2. Note that the results are given with **two** digits after the decimal point. As it has to be, you may say. But you may have already found out that doing plain arithmetic on your computer rarely produces results with just two digits after the decimal point. If you haven't then try this: PRINT 10/3 and press RETURN.

Any arithmetical procedure on a computer that is likely to produce more digits after the decimal point, will indeed result in many digits. Often too many, such as in cases where money is involved. Financial calculations require only two digits after the decimal point but, more importantly, the figure has to be **rounded**! A calculation producing, for example, 45.6789 must be displayed as 45.68. Your Commodore-64 does not have a special function that rounds figures. However, you can create one very easily, thanks to the DEF FN combination offered by BASIC. Read about it in your User Manual, but the book 'Discover your Commodore 64' gives a more elaborate explanation. NOTE also that the Appendix of this book offers a rounding function that will do precisely what we need in this project.

Hence the combination DEF FN literally means DEFine FuNction. After DEF FN follows the name of the function (two letters) and a dummy variable, used in the arithmetical funtion that has to be defined. Thus the function that rounds a figure to two digits after the decimal point can be

written this way:

DEF FN RD(X) = INT (X * 1ØØ + .5)/1ØØ

Where DEF and FN define the function,

RD is the name of the function (RD stands for RounDing, but could be any other name);

X is the **dummy** variable, which means that the function will work with any other variable than X.

INT stands for the BASIC function INT(X) that returns the integer of a real number;

The statement INT (X * 1ØØ + .5)/1ØØ is the BASIC translation of the formula

$$\frac{\text{INT}\,(X * 1ØØ + Ø.5)}{1ØØ}$$

QUESTION: can you find out for yourself why the above formula rounds a figure to two digits after the decimal point?

Once the function has been defined, any variable in the program containing a real number that needs rounding, can now be treated with this function. For example, when variable A contains 45.6789, the following BASIC statement:

A = FN RD (A): PRINT A

will produce 45.68 as the contents of A, displayed on the screen.

RULE: a DEF FN function must always be defined at the beginning of the program, that is in the initialisation routine. This is why this discussion was connected with the initialisation routine.

You are now ready to write it, including the defined rounding function.

```
1ØØ  REM *** INITIALISATION ROUTINE ***
11Ø  REM (CLEAR SCREEN)
12Ø  REM (HERE PRINT TITLE OF PROGRAM)
13Ø  REM (PRINT TWO BLANK LINES)
14Ø  REM (DEFINE ROUNDING FUNCTION)
15Ø  RETURN
```

5.5 Testing the Initialisation Routine

As usual the initialisation is so simple that not much can go wrong. However, now that you have entered a defined function, syntax errors may occur in this specific program line. But, take particular notice of the following facts:

1. A syntax error will occur when, for example, you write DEF F RD(X), or DEF FN RD(X, or anything else which does not follow the correct syntax of this function. The computer will report such an error.
2. The statement at the **left** side of the equals **sign** (=) may be fully correct, but there may be something wrong on the **right** side: a logic error, or a syntax error. **In that case the syntax error will be reported elsewhere in the program, and not in the line where the function has been defined!**

Example:

```
1Ø  DEF FN RD(X) = INT(X * 1ØØ + .5)/1ØØ)
2Ø  A = 45.6789
3Ø  PRINT FN RD (A)
```

This little program will report a syntax error in line 3Ø. Why? Simply because the function has been incorrectly defined. Consider line 1Ø and see that there is one bracket too many (at the end of the expression, after 1ØØ)

Hence the syntax error in 1Ø will be reported in 3Ø, because it is there that the defined function is put to the test.

When there is a logic error on the right side of the equals sign in the defined function, the program will probably work but come up with unwanted results!

5.6 The input data

Writing the input routine will be very easy. It will display the screen prompts, and the only thing the user has to do is to enter the required data: (1) borrowed amount, (2) the duration of the mortgage and (3) the annual interest rate. These are all numerics, hence need only numeric variables. How shall we call them? As you will see later on, there exists a well defined formula for the calculation of mortgage repayments. In this formula the borrowed amount is given the letter P (for 'Principal'), the duration of the mortgage is given the letter N (for 'N' years, a widely used expression when an unknown number must be specified in a formula), and the interest rate is represented by the letter I. There is

every reason to maintain these letters as the appropriate variables in this program. Hence P will contain the borrowed amount, N the number of years the mortgage is effective, and I the interest rate.

By the way, reading computer magazines and computer books may reveal a word to you that you may not have encountered before. It is the word 'parameter', which often refers to data to be input and processed. Hence P, N and I contain the 'parameters' of this mortgage repayment program. It is a word that I will not use very much, because it is one of the least comprehensible words in computer jargon.

Writing the input routine, beginning at line 200, will be a quick job:

```
200 REM *** INPUT ROUTINE ***
210 REM (INPUT BORROWED AMOUNT   — STORE IN P)
220 REM (INPUT DURATION MORTGAGE — STORE IN N)
230 REM (INPUT INTEREST RATE     — STORE IN I)
240 REM (DIVIDE INTEREST RATE BY 100)
250 RETURN
```

NOTE the warning in paragraph 5.7, under 'The interest rate'!

5.7 Perform the calculations

BASIC is a computer language derived from the oldest 'high level' computer langauge, FORTRAN. This anagram stands for FORmula TRANslation, hence it was primarily designed for mathematical applications.

Thus FORTRAN allows mathematical formulae to be entered into the computer in almost identical form. The designers of BASIC showed great foresight with their decision to adopt a similar way of handling mathematical functions.

Therefore the translation of seemingly complex mathematical formulae into BASIC program lines is by no means a difficult task.

This is the mortgage repayment formula:

$$R = \frac{P * I * (1 + I)^N}{(1 + 1)^N - 1}$$

This formula also works when you write it this way:

$$R = (P * I * (1 + I)^N)/((1+I)^N - 1)$$

In this form it may be translated into BASIC, except for the little problem of raising to the N-th power. To achieve that, you must use the standing arrow (↑). Thus in BASIC the program line that does the entire calculation in one go looks as follows:

R = (P * I * (1 + I) ↑ N) / ((1 + I) ↑ N – 1)

If you find this too hard, there is nothing against the alternative of splitting up the formula.

You will find $(1+I)^N$ in both parts of the formula. For your convenience, call this 'A'. Hence A = (1+I) ↑ N. Now work out the formula in BASIC:

A = (1+I) ↑ N: B = P * I * A: C = A – 1: R = B/C

The result will be exactly the same but, of course, the formula-type BASIC program line is considered to be more elegant.

The interest rate. You are now ready to enter the BASIC representation of the mortgage repayment formula but, before you do so, we still have to consider an important aspect of the interest rate. The formula as is will not produce the proper result when you leave the interest rate unchanged. That is, it is not right to work with say 11.5 when what is meant is 11.5 per cent. Thus in order to get the correct result you must divide the interest rate by 1ØØ, hence the BASIC program statement will become I = I/1ØØ. Where should this be placed? **IN THE INPUT ROUTINE!** Placing it in the calculation routine will cause a 'beautiful' logic error, when the program is complete, i.e. including the options routine.

It really is interesting to see what happens when you insert the I = I/1ØØ statement at the start of the calculation routine instead of at the end of the input routine. Because if you then enter, via the options routine, a fresh borrowed amount, you will discover that your monthly repayments become less and less, and that at a certain moment the computer will report a "division by zero error". Try to find out the how and why of this logic error.

Monthly payments. This is simple. MP = R/12 (MP stands, of course, for Monthly Payment). Round it to two digits after the decimal point: MP = FNRD(R/12).

Weekly Payments. This is a bit more involved. One would think that WP = R/52 would do. That is not so, because then it is assumed that a year has exactly 52 weeks. But as a year has not (7 times 52 =) 364 but 365 days, we usually are a day short. With a leap year we are two days short.

It is therefore impossible to find an exact weekly amount. We can only

approximate it: divide R by (365/7 = 52.15). Hence a BASIC program line will be WP = FNRD(R/52.15).

Total Payments. TP = FNRD(R * N).

You are now ready to enter the calculation routine.

```
300 REM *** CALCULATION ROUTINE ***
310 REM (CALCULATE "R" — (MORTGAGE FORMULA))
320 REM (CALCULATE MONTHLY PAYMENT AND ROUND)
330 REM (CALCULATE APPROX. WEEKLY PAYMENTS AND ROUND)
340 REM (CALCULATE TOTAL PAYMENT OVER N YEARS AND
    ROUND)
350 RETURN
```

5.8 Display the results

The display routine starts at line 400 and won't give you any problems. To obtain a neat lay-out on the screen, I strongly suggest you plan the format of your output on a screen map.

NOTE also the line displaying the annual interest rate in %. What are you required to do to place the correct rate on the screen? (if you can't find the answer, refer to the appendix).

```
BORROWED AMOUNT   (IN $$):  XXXXX
DURATION OF MORTGAGE(YRS):  XX
ANNUAL INTEREST RATE (%):  XX

MONTHLY REPAYMENTS:  XXX
APPRX. WEEKLY REPAYMENTS:  XX
TOTAL OVER XX YEARS:  XXXXX

(I)NTEREST RATE
(B)ORROWED AMOUNT
(D)URATION
(E)XIT

PRESS APPROPRIATE LETTER FOR VARIATIONS
```

```
400 REM *** DISPLAY ROUTINE ***
410 REM (PRINT BORROWED AMOUNT — STORED IN P)
420 REM (PRINT DURATION MORTG.  — STORED IN N
430 REM (PRINT INTEREST RATE (%) — STORED IN I —
                                 MULTIPLY BY 100)
440 REM (PRINT MONTHLY REPAYM.  — STORED IN MP)
450 REM (PRINT WEEKLY REPAYM.   — STORED IN WP)
460 REM (PRINT TOTAL PAYMENTS   — STORED IN TP —
                                 NUMBER OF YEARS IN N)
470 REM (PRINT A BLANK LINE)
480 RETURN
```

5.9 Testing the Calc. and Display routines

Apart from the usual bugs caused by typing errors, a source of problems may be the line representing the mortgage repayment formula. It contains many brackets, and quite often brackets are forgotten or there are too many of them. Also note here the 'I' and the digit '1' — do not confuse them. And of course there may be a syntax error, reportedly in the calculation routine, but it will be located in the define function line of the initialisation routine.

The first diagram of this chapter not only gives you the input data (or 'parameters') but also the calculation results. When your program, if 'fed' with the same data, displays the same results, it would seem to be working correctly. If the results are not the same, there is a logic error somewhere and you must figure out where the calculations went wrong. Arithmetic with the computer requires that you adhere to the rules of mathematical hierarchy. What is between parenthises (brackets) comes first, exponentiation (the standing arrow) comes next, then multiplication and division, and last on the list are addition and subtraction.

If, for example, you had written:

$$R = P \times I \times (1 + I) \uparrow N/(1 + I) - 1$$

the program would have worked, but the results would have been wrong! NOTE: the part of the division on the left side of the division slash can be written without outer brackets; the right side part **must** have outer brackets! Try to find out why.

5.10 Developing the Options

The options routine starting at line 500 is very straightforward. The first few lines contain print statements (see diagram 23) prompting the user

to press a key which causes the program flow to go to an appropriate lower level subroutine. There are three of those subroutines, each in fact prompting the user to enter an alternative 'parameter', i.e. interest rate, borrowed amount or duration. The exit prompt is meant to end the program. NOTE that after the assessment of each of those subroutines, a flag must be set causing the program flow to go to the calculation routine via the mainline. The exit prompt clears the flag. The three alternative entry subroutines are addressed via the program lines 1000, 2000 and 3000.

```
500 REM *** OPTIONS ROUTINE ***
510 REM (PRINT BLANK LINE)
520 PRINT"(I)NTEREST RATE"
530 PRINT"(B)ORROWED AMOUNT"
540 PRINT"(D)URATION"
550 PRINT"(E)XIT"
560 PRINT
570 REM (HERE PRINT SCREEN PROMPT — SEE DIAGRAM 23)
580 REM (HERE WRITE "GET. . ." STATEMENT)
590 IF X$ = "I" THEN GOSUB 1000:GOTO 640
600 REM (CONDITIONAL TEST FOR B — SUBR. 2000, THEN TO 640)
610 REM (CONDITIONAL TEST FOR D — SUBR. 3000, THEN TO 640)
620 REM (CONDITIONAL TEST FOR E — CLEAR FLAG HERE AND GO
    TO EXIT)
630 REM (PROGRAM FLOW RETURNS TO 580 IF DROPPING
    THROUGH)
640 REM (SET FLAG HERE)
650 RETURN
```

5.11 One of the Alternative Entry routines

Here is one of the alternative entry routines, fully written. The two others are similar, so you can write them yourself.

QUESTION: why the 'clear screen' lines? You find here again the program statement I = I/100; do the others need a similar arithmetical line?

```
1000 REM *** ALTERNATIVE INTEREST ROUTINE ***
1010 PRINT " ": REM — CLEAR SCREEN
1020 INPUT "ENTER ALTERNATIVE INTEREST RATE";I
1030 I = I / 100
1040 PRINT" "
1050 RETURN
```

5.12 The final test-RUN

You just finished the program, and when you give it the final test-RUN, not only should you get the proper results on the screen but you should also be able to obtain different results by entering alternative data via the options routine. Here are a few screens with alternative results. If yours are the same, then we may assume that your program is correct. But beware! One of 'Murphy's laws of computer programming' states that 'the worst bug is found when the program has been in business for many months'.

```
BORROWED AMOUNT (IN $$)      : 3ØØØØ
DURATION OF MORTGAGE (YRS) : 25
ANNUAL INTEREST RATE (%)     : 11.5

MONTHLY REPAYMENTS           : 3Ø7.75
APPRX. WEEKLY REPAYMENTS     : 7Ø.81
TOTAL OVER 25 YEARS          : 92323.52

(I)NTEREST RATE
(B)ORROWED AMOUNT
(D)URATION
(E)XIT

PRESS APPROPRIATE LETTER FOR VARIATIONS


BORROWED AMOUNT (IN $$)      : 3ØØØØ
DURATION OF MORTGAGE (YRS) : 25
ANNUAL INTEREST RATE (%)     : 15

MONTHLY REPAYMENTS           : 386.75
APPRX. WEEKLY REPAYMENTS     : 88.99
TOTAL OVER 25 YEARS          : 116Ø24.55

(I)NTEREST RATE
(B)ORROWED AMOUNT
(D)URATION
(E)XIT

PRESS APPROPRIATE LETTER FOR VARIATIONS
```

5.13 Extensions

A very practical program like this lends itself excellently to useful extensions. For example, you can add a subroutine that prints the contents of the screen on paper (if you possess a printer). A subroutine printing the entire screen is normally known as a Screen Dump. Writing screen dump routines is something for highly advanced programmers who know all the ins and outs of the computer, and therefore a subject beyond the scope of this book. However, you will find a screen dump routine in the Appendix. If you want to add it, simply insert another line into the option routine, prompting the user to press, for example, the 'P' key, and consequently the contents of the screen will be 'dumped' on paper. By the way, programmers often talk of 'hard copy' when they refer to printed (on paper) output.

Another thing to consider is the deplorable fact that interest rates are not fixed. You may create subroutines allowing for such variations, enabling you, for example, to find out how much extra you have to pay, after so and so many years of repayments, and how much more it may cost you in the long run. You may also allow for inflation, tax-rebates and so on.

CHAPTER 6
A Word Guessing game

6.1 A small change in tuition method

Assuming that by now you have some confidence in programming, after successful completion of all four previous projects, it is time to adopt a somewhat different approach. You will still be provided with the framework of this new project but much more will be asked of you. It is comparable to a true-to-life situation at a software firm where the project leader assigns a job to one of his trainee programmers. The trainee receives a job description, and the project leader tells him in broad terms how to design the program. Hence the trainee receives a description of the mainline and of the most relevant subroutines. He also receives hints in regard to special keywords that may be helpful in resolving the various programming problems.

This is a true challenge, and as such highly rewarding when you manage to complete it successfully.

6.2 The seed idea

One of the reasons why people buy a home computer is its usefulness as an educational tool. Children soon feel at home with a computer and appear to be very willing to play games with them, particularly when those games are of an educational nature. This may inspire you to develop some simple educational programs on your own, for use by your children. Word guessing is such a game. The user — your child in this instance — is asked by the computer to choose from a variety of categories, for example 'Farm animals', 'Zoo animals', 'Your toys', and so on. Every category is indicated by a number, and the child must press one to initiate and assess the specified group.

The program finds the specified group and picks — **at random** (!) — a word, displays its length (it says 'this word has . . . letters'), the first letter and the last letter. The child is then asked by the computer to think and eventually type in the word it guesses to be the correct one. Consequently the program tries to match the word just input by the child with the word it has randomly selected from the previously specified category. If the guessed word matches, then the computer may display an encouraging reaction, something like 'Aren't you smart, you guessed it right'.

If not correct, the program asks to try again, yes or no. With yes, the child gets another chance to enter a guess. With no, the child goes back to the input, where another category (or the same) cand be selected.

6.3 Handling the database — in general terms

As in previous projects, this one is centred around a database, containing several words. Somehow, the category to which a word belongs must be identified in the record (as you know, the 'record' is the item placed on the DATA line). For example, the category 'Farm animals' may get the number 1 as its record key. Thus a DATA line may look something like 5Ø1Ø DATA "1-HORSE". The category 'Zoo Animals' may get as record key number "2", as in 511Ø DATA "2-GORILLA"; and so on.

When a category is selected from the input, the program starts to READ the items from the database but will only select those words matching the category number just entered. The question arises, 'how to place these selected words into easy accessible string variables?' We are not only faced with that little problem but also with the fact that, later on, the program must select — **at random** — a word from one of these string variables. Ordinary string variables are not very suitable in a case like this. But 'subscripted variables', or **arrays**, are created, as it were, for this type of work: storing data belonging to the same category. Subscripted variables are of the type A(1), A(2), A(3), etc. — see your Users Manual, or 'Discover your Commodore-64'. (**NOTE: do not confuse with A1, A2, A3, A4, etc**. These are not subscripted variables!)

However, an array can only be used when it has been 'dimensioned'. Here the BASIC keyword DIM brings salvation (in the initialisation routine, see later). The program must select — at random — a word from the array. This suggests a routine generating random numbers, and is therefore called a 'Random Generator'. Generating random numbers can only be done with the help of the BASIC function RND(X).

Finally, the selected word needs to be manipulated:

its length will be obtained with the BASIC function LEN(X$);
the leftmost character with the function LEFT$(X$, 1);
the rightmost character with the function RIGHT$(X$, 1).

6.4 The overall set-up

1. Mainline as first block.

2. Initialisation Routine — contains explanatory text, and DIM statement.

3. Input Routine — displays categories, and requires entry of record key.

4. Read Routine — reads the data and matches them with the record key; if there is a match the routine refers to a lower level subroutine that retrieves the word without record key and places them in a subscripted string variable.

5. Random Generator — generates a random number, with which a word will be retrieved from the array of subscripted string variables.

6. Display and Guess Routine — displays the length of the word, the first and the last letter, and waits for the input of the guess word. Matching of the guess word with the randomly selected word follows.

7. Another Go Routine.

8. Third level subroutine retrieving the record (= the word) without record key, and filling an 'array'.

9. Database.

6.5 The Mainline

Below is the full mainline. NOTE that considerable space has been reserved for every second-level subroutine.

```
1Ø REM *** WORD GUESS ***
15 :
2Ø REM * MAINLINE *
3Ø GOSUB 1ØØ          : REM — TO INITIALISATION ROUTINE
4Ø GOSUB 4ØØ          : REM — TO INPUT ROUTINE
```

```
5Ø GOSUB 6ØØ        : REM — TO READ ROUTINE
6Ø GOSUB 8ØØ        : REM — TO RANDOM GENERATOR
7Ø GOSUB 1ØØØ       : REM — TO DISPLAY AND GUESS ROUTINE
8Ø GOSUB 12ØØ       : REM — TO ANOTHER GO ROUTINE
9Ø IF FL=1 THEN 4Ø  : REM — FLAG TEST
95 END              : REM — PROGRAM ENDS HERE
```

6.6 The Initialisation Routine

Apart from clearing the screen and then displaying the title of the program, this routine contains a DIM statement, that 'dimensions' the length of the array (of subscripted variables) that will store the words of the selected category. I suggest "SC$()" as the variable name (**S**elected **C**ategory), and the DIM statement then becomes DIM SC$(15) if you want to go to a maximum of fifteen words. (You can go up to a maximum of 255!) (N.B. names for all other variables are suggested in paragraph 6.12). If you want to use the initialisation routine for the display of explanatory text, then the GET statement will be handy to prevent the program flow leaving the subroutine and thus clearing the screen.

Use the format:

(line number) GET X$: IF X$ = " " THEN (line number)

Pressing any key will force the program flow to leave the Initialisation Routine. N.B. It is then necessary to place a screen prompt, "PRESS ANY KEY TO CONTINUE"

```
1ØØ REM * INITIALISATION ROUTINE *
11Ø REM (CLEAR SCREEN)
12Ø REM (DIMENSION SELECTED CATEGORY STRING)
13Ø REM (FROM HERE PLACE ON REM'S IDENTIFYING VARIABLES
    OF THIS PROGRAM)
2ØØ REM (FROM HERE ON PRINT EXPLANATORY TEXT DO
    NOT FORGET TO INCLUDE BLANK LINES)
25Ø REM (HERE PLACE GET STATEMENT)
26Ø REM (CLEAR SCREEN)
27Ø RETURN
```

6.7 The Input Routine

This one speaks for itself. Note that you have a choice of two ways to get out of the input routine:

1. Using the input command — entering a number and pressing RETURN.

2. Using the GET statement. This is far more elegant.

In any case, be aware of the possibility that the user may accidentally type a different number to those displayed on the screen.

The routine must allow for this. A line like:

IF CT <1 OR CT>6 THEN (clear screen): GOTO (entry point of routine) will take care of this.

NOTE: The variable CT (for CaTegory) is suggested here.

```
4ØØ REM * INPUT ROUTINE *
41Ø PRINT "      MAKE YOUR CHOICE"
42Ø PRINT "(PRESS ONE OF THE NUMBERS)"
43Ø PRINT:PRINT"FARM ANIMALS      (1)"
44Ø REM (FILL OUT THE REMAINDER)
5ØØ INPUT CT : REM (SEE TEXT — A GET STATEMENT WILL BE
    BETTER)
51Ø RETURN
```

6.8 The Read Routine

No great problems here, except that retrieving the record key for match comparison with the category number in CT needs special consideration. The LEFT$(X$) function enables you to get out the record key, but although it is a numeric, the computer still sees it as a string. And, as you know, the contents of a string variable cannot be compared with the contents of a numeric variable. There is a simple way to get around this problem. Use the function VAL(X$), which will return the numeric value, and thus is comparable with the contents of CT.

IMPORTANT: take special notice of that tiny statement in line 61Ø, I = Ø. This 'setting to zero' of the variable 'I', to be used in subroutine 2ØØØ, is crucial in the correct functioning of the program. If you leave it out, the program may RUN correctly for the first time (or maybe even a few times when you made wrong guesses), but it will inevitably 'crash'. Try to find out for yourself the how and why.

```
6ØØ REM * READ ROUTINE *
61Ø REM (SET INCREMENT VARIABLE I TO ZERO)
62Ø REM (READ STATEMENT)
63Ø REM (CONDITIONAL TEST — IF THE NUMERICAL VALUE OF
    THE RECORD KEY EQUALS CT THEN GO TO SUBR 2ØØØ)
    SUBR 2ØØØ)
```

```
64Ø REM (CONDITIONAL TEST OF EOF)
65Ø GOTO 62Ø
66Ø RETURN
```

6.9 Subroutine 2ØØØ 'fills the array'

The read routine points to a third-level subroutine, starting at 2ØØØ and is called an 'array filler'. It determines the length of the record, retrieved from the database and decreased with the length of the record key. A RIGHT$ statement then picks out the actual word from the record, using the length of the word as the argument. The word is then placed into the subscripted variable. The subscripted variable is indicated by the increment variable "I": SC$(I). The contents of I determines in what subscripted variable of the array the word is placed. The crux of the matter is that the contents of "I" must be increased every time that an appropriate record is READ. If not, only one subscripted variable of the array will continuously be filled, and refilled. When the contents of I are increased by 1 every time a record is READ, the entire array will be filled out, e.g.: SC$(1) = "Elephant", SC$(2) = "Gorilla", SC$(3) = "Pelican", and so on.

When the entire array has been filled, the increment variable I must be set to zero. The best place to do that is at the beginning of the READ routine.

```
2ØØØ REM * ARRAY FILLER *
2Ø1Ø REM (INCREMENT VARIABLE I WITH 1)
2Ø2Ø REM (LENGTH OF WORD EQUALS LENGTH MINUS 2(= LW) )
2Ø3Ø SC$(I) = RIGHT$(DA$,LW)
2Ø4Ø RETURN
```

6.10 Generating Random Numbers

A randomly selected number is needed to retrieve at random a word from the SC$() array. For example, if you have 'dimensioned' the array up to four, then its contents may be as follows:

SC$(1) = "ELEPHANT", SC$(2) = "BEAR", SC$(3) = "GORILLA", SC$(4) = "PELICAN".

If the random generator of this project produces the number 3, for example, the program is written in such a way that it displays the contents of SC$(3). Consider this: the random number is placed in a variable named RN. The sequence of program statements may then become:

RN = (random generator) : PRINT SC$(RN)

Take special notice of RN between the brackets!

A random number is generated by the BASIC function RND(X), where X can be any number, because what is generated under all circumstances is something, at random, between zero and 1. Just try PRINT RND(1) and press RETURN. What you get, however, is not usable in this form. The following line generates randomly a whole number between Ø and 11 (hence does not include Ø and 11):

RN = INT(RND(1) * 1Ø) + 1

NOTE: 2Ø instead of 1Ø generates a number between Ø and 21; 3Ø between Ø and 31, etc, so that will be the core of this random generator routine. The rest of it can be devoted to the retrieval of the word from the array, establishing its lengths, and leftmost and rightmost letters of the word.

```
8ØØ REM *RANDOM GENERATOR *
81Ø REM (GENERATE RANDOM NUMBER BETWEEN Ø and 1Ø)
82Ø REM (RETRIEVE WORD FROM ARRAY)
83Ø REM (ESTABLISH LENGTH OF WORD (=LN) )
84Ø REM (ESTABLISH FIRST LETTER OF WORD)
85Ø REM (ESTABLISH LAST LETTER OF WORD)
86Ø RETURN
```

6.11 Display and Guess

The display and guess routine, starting at line 1ØØØ, is easy to write. The most important line will be the one where the guess word is matched against the randomly selected word from the chosen category. It needs only one additional test: if there is a match then the computer will display a message telling the user not only that the guess was correct, but giving an encouraging message as well. Consequently the program flow branches to the exit point of the routine. If there is no match, the program flow 'drops through' and a "sorry, etc." message may be displayed on the screen. Next in line is a screen prompt asking the user whether he/she wants another try. So here is the outline of the display and guess routine, to be filled out by you:

```
1ØØØ REM * DISPLAY AND GUESS ROUTINE *
1Ø1Ø REM (CLEAR SCREEN)
1Ø2Ø PRINT "THE WORD IS";LN;"LETTERS LONG"
1Ø3Ø REM (PRINT A BLANK LINE)
1Ø4Ø PRINT"THE FIRST LETTER IS ";LL$
1Ø5Ø REM (PRINT THE LAST LETTER)
```

```
1Ø6Ø REM (PRINT TWO BLANK LINES)
1Ø7Ø REM (PR!NT PROMPT TO ENTER GUESS WORD)
1Ø8Ø REM (HERE INPUT OF GUESS WORD)
1Ø9Ø REM (CONDITIONAL TEST — MATCH GUESS WORD WITH
     WORD)
11ØØ REM (PRINT "SORRY,ETC", THEN BLANK LINE)
111Ø REM (PRINT SCREEN PROMPT "TRY AGAIN? ETC)
112Ø REM (INPUT YES OR NO)
113Ø REM (CONDITIONAL TEST YES — IF TRUE THEN 1Ø1Ø)
114Ø REM (CONDITIONAL TEST NO — IF TRUE THEN EXIT)
115Ø GOTO 111Ø
116Ø RETURN
```

6.12 "Another Go" and some suggestions

There is nothing new about the another go routine — simply copy it from your previous projects.

You are now ready to write this program. To conclude, here are some extra suggestions.

Start the database high up in the line numbers, for example 5ØØØ, and end it with the usual 9999 "EOF". This leaves you not only enough room to add as many categories and words you like, but there is also room for program extensions. Later on you may design routines that enable the program to partially match the two words, and tell the user something like "you are very close — four of the six letters are correct". Another routine may count the guesses, and provide the user with rewards in points, like 1ØØ points for a guess in one go, 5Ø for two guesses, and so on.

Keep in mind that a game like this may become 'boring' for the youngsters when they learn to know by heart all the words of the database. In that case you won't have any trouble in updating the program — simply delete the old categories and introduce new ones.

Finally, here is a list of suggested string and numerical variables to be used in this project:

SC$()	Selected Category array
CT	Category number
DA$	Record of DATA file
RN	Random number
WR$	Randomly selected word from category array

LN	Length of word (placed in WR$)
LW	Length of word in Array Filler Routine
LL$	Leftmost letter of word
RL$	Rightmost letter of word
GW$	Guess word, to be matched against contents of WR$
TR$	Contains Y or N of 'Try again?'
I	Increment variable I

NOTE: It is a good idea to identify these variables in the form of REM statements in the initialisation routine.

6.13 To conclude this chapter

We have not discussed test-RUNs and possible bugs, because I assume that by now you will be perfectly able to locate them yourself. If not, the appendix will offer you some clues.

The next and last project in this book will even be more of a challenge — again it uses a random generator, but it also introduces the FOR-NEXT loop, and some computer graphics.

CHAPTER 7
Rolling three dices

7.1 The seed idea

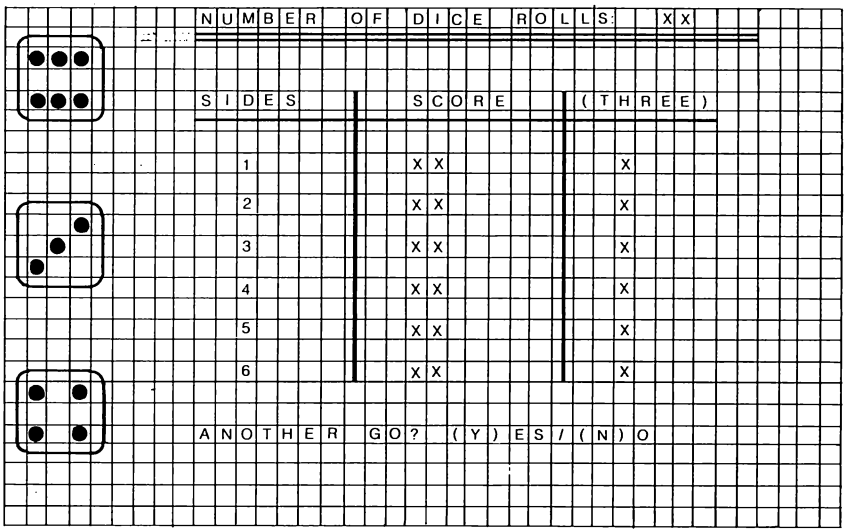
'Dice Rollers' feature in many books about BASIC programming and usually they have all the characteristics of a simple game. This project aims less at the development of a game and more at a program that counts the results of subsequent dice rolls and prints them in an attractive table form on the screen. However, the same programming techniques that are developed in this project can be used in case you want to create a dice roller game on your own.

The project as is must create a program that, at the touch of a key, chooses **at random** three numbers between Ø and 7 (i.e. Ø and 7 are excluded) and, depending on the numbers selected, three dices are displayed on the screen, each of them confirming the randomly chosen digit. Thus if the random generator selects 2, the result is the image of a dice showing the side with two dots; in the case of say 4, the four dot side is displayed, and so on. This points to the application of Commodore's computer graphics.

When the three dices are displayed, for example at the left side of the screen, the program will evaluate the results of the roll, and display them on the right side in a table. A special feature will be the count and display of the occasional roll where three of the same dice side come up.

The next diagram, depicting a possbile lay-out of the display on a screen grid form, gives you an idea of how to set up your output format.

There are a few BASIC statements and commands that you haven't encountered in the previous projects but you will be introduced to them in the present one.



7.2 An overview of the routines

Apart from the mainline, the initialisation and another go routines, all of them easy for you to develop, there are three routines requiring special attention. The first of them is the 'Dice Roller'. It contains the random generator that produces the three randomly selected numbers between Ø and 7 (Ø and 7 excluded). It is in this routine that you will apply a so-called FOR-NEXT loop, whilst you will again use the ON-GOSUB combination introduced in project 1.

The 'ON-test' of the result of the random generator will determine to which 'dice subroutine' the program will flow.

The next routine tests whether the one roll of three dices comes up with three of the same side, e.g. three sixes, three ones, three fours, and so on. If so, it will be registered and placed in a variable whose contents will be increased by one unit, every time the same side of the dice comes up three times in the one roll. Take special notice of the fact that a separate count must be provided for all six sides.

This routine introduces you also to the BASIC statement "AND", a so-called 'logical operator'.

The next routine is the one which displays the results in tabular form. As this may be too unfamiliar to you, the entire routine is given, together with some explanatory comments. It is then only a matter of typing it in correctly.

7.3 Suggested variables

This project may contain three arrays, because some 'parameters' are of a similar nature:

1. The random generator produces the numbers 1, 2, 3, 4, 5 and 6, each representing a side of the dice. Hence they can be stored in a subscripted variable DI(). As there are three rolls in one go, producing three different numbers (though sometimes all three results are the same) we must set aside three variables for this array: DI(1), DI(2) and DI(3).
2. Six sides are randomly accessed, and each of them requires a variable, storing its score. We can set aside for this six, subscripted variables, for example SC(1), SC(2), SC(3), SC(4), SC(5) and SC(6).
3. Then there are the occasional hits of three-the-same-side in one roll. We can set aside for them the array: TH(1), TH(2), TH(3), TH(4), TH(5) and TH(6).

Suggestions for other variables:

I as the loop variable (in the FOR-NEXT loop);

RC as the Roll Counter;

Y as the side evaluator.

IMPORTANT: the Commodore -64 has the provision that arrays smaller than ten subscripted variables do **not** need to be DIMensioned. However, we advise you to DIMension your arrays under all circumstances in order to remain aware of the fact that arrays are used in the program. Hence reserve a line in your initialisation routine for the DIMensioning of the previously described arrays.

7.4 The Mainline

This mainline is almost identical to all others in this book.

```
10 REM *** DICE ROLLER ***
15 :
20 REM * MAINLINE *
30 GOSUB 100          : REM — TO INITIALISATION
```

```
4Ø  GOSUB 2ØØ        : REM — TO RANDOM GEN. & DICE ROLLER
5Ø  GOSUB 3ØØ        : REM — TO EVALUATION FOR THREE
6Ø  GOSUB 4ØØ        : REM — TO DISPLAY EVALUATION
7Ø  GOSUB 5ØØ        : REM — TO ANOTHER GO
8Ø  IF FL=1 THEN 4Ø   : REM — FLAG TEST
9Ø  END               : REM — END OF PROGRAM
```

7.5 The Initialisation Routine

```
1ØØ  REM *INITIALISATION *
11Ø  REM (CLEAR SCREEN)
12Ø  REM (DIMENSION YOUR ARRAYS (DIM
     DI(3),SC(6) ETC) )
13Ø  REM (YOU MAY PRINT FROM HERE ON SOME)
14Ø  REM (EXPLANATORY NOTES FOR THE USER)
15Ø  REM (PRINT PROMPT TO PRESS ANY KEY TO START)
16Ø  REM (PLACE HERE YOUR GET . . . STATEMENT)
17Ø  REM (CLEAR SCREEN)
18Ø  RETURN
```

7.6 The Dice Roller

The core of this routine is, of course, the random generator. You have worked with that one before, so writing one now that generates the numbers from 1 to 6 will be easy. The result of the random generation must be stored in the subscribed variable DI(I). NOTE the letter 'I', not the digit '1'! Why such a format? This leads us to the FOR-NEXT loop.

The sequence FOR (follows 'loop' variable) = (starting number) TO (end number), follows a program routine, and then NEXT ('loop' variable), initiates a **limited** loop.

For example this simple routine:

```
1Ø  FOR J = 1 TO 1ØØ
2Ø  PRINT "*";:
3Ø  NEXT J
```

prints horizontally a hundred asteriskes on the screen. In line 1Ø the loop variable J is filled with the digit 1, in line 2Ø an asterisk is printed, in line 3Ø, however, the content of J is **tested**. If J is not larger than 1ØØ, the program flow goes back to 1Ø whereas the loop variable J is incremented with one unit! If larger than 1ØØ, the program flow leaves the loop and 'drops through' to the next line. A more elaborate explanation of this very useful set of BASIC statements can be found in 'Discover Your Commodore -64'. Also consult your Users Manual. What does it do in the

dice roller routine? Quite simply this: FOR I = 1 TO 3 initialises and maintains a loop of three cycles. During each cycle a random number is generated and stored in DI(I)!

But then consider the fact of "I" being the subscript of DI(I). When the contents of I increases one unit every cycle, we effectively create an array: DI(1), DI(2) and DI(3). If the first cycle produces as the random number say 5, then DI(1) contains 5; the second cycle generates say 3 and stores 3 in DI(2), and the last cycle generates say 1 and stores 1 in DI(3). Line 23Ø contains the ON-GOSUB statement that, on the basis of the contents of DI(I), determines to which 'dice subroutine' the program will flow. Of course, you may use the alternative of six conditional tests:

```
IF DI(I) = 1 THEN GOSUB 1ØØØ
IF DI(I) = 2 THEN GOSUB 11ØØ
```

and so on, but the ON-GOSUB combination is more elegant as it requires only one program line.

Finally, consider line 25Ø. It contains the counting variable RC (which stands for roll count), to be used in the display routine. It must have the capacity to increment the contents by one unit every time the program generates a three fold dice roll.

```
2ØØ  REM * DICE ROLLER *
21Ø  REM (PLACE HERE FOR . .TO STATEMENT)
22Ø  REM (RANDOM GENERATOR)
23Ø  REM (ON . . . . GOSUB . . .)
24Ø  REM (NEXT (LOOP VARIABLE) HERE)
25Ø  REM (ROLL COUNTER HERE)
26Ø  RETURN
```

7.7 The Dice Display routines

Consider the fully developed routine below:

```
13ØØ  PRINT"  ,---,  "
131Ø  PRINT" |●  ●| "
132Ø  PRINT" |   | "
133Ø  PRINT" |●  ●| "
134Ø  PRINT"  `---'  "
135Ø  SC(4)=SC(4)+1
136Ø  PRINT
137Ø  RETURN
138Ø  :
139Ø  :
```

The simplicity of this routine needs no elaboration. It will be just a matter of minutes for you to type the other dice routines. Start them at the following lines: 1ØØØ (for side 1), 11ØØ (for side 2), etc: 12ØØ, 13ØØ, 14ØØ and 15ØØ. Take special notice of the SC() subscripted variables — they count the score of each side!

Insofaras the special graphics comprising the dice image are concerned, it is up to you to find them on your Commodore -64 keyboard. You shall have to press either the 'Commodore-key' or the 'Shift-key' simultaneously with other keys to have them displayed.

7.8 Count of three of the same sides in one hit

This is much easier than you may have thought. The essence of the programming problem here is that the contents of all three members of the DI() array must be compared with each other, and a count is made only if all of them contain equal numbers. The 'Logical Operator' **AND** takes care of this, because what we want is that DI(1) **AND** DI(2) **AND** DI(3) have the same contents to meet the condition. Hence; the program line may become something like this: IF DI(1) = DI(2) AND DI(3) = DI(2) THEN . . . (to be filled out).

Now if, for example, all three numbers of the array contain the number three, then it is side three which scores one hit of three of that one roll. We may use the number of the side as the argument of a subscripted variable that contains the count of the times there is a hit of three equal numbers in one roll. Therefore, consider Y = DI(1): TH(Y) = TH(Y) + 1. This will do the job perfectly when it is placed after THEN of the AND-AND condition (see above). So as you will see below, the entire evaluation routine contains one line only, except for the entry and exit lines.

```
3ØØ REM * EVALUATION OF THREE *
31Ø REM (IF-AND-AND-THEN . . . .)
32Ø RETURN
```

QUESTION: is there any reason to replace Y = DI(1) by Y = DI(2) or Y = DI(3)?

7.9 Displaying the table of results

As promised the entire display routine is provided below. You only need to type it in. Take particular notice of the use of the Commodore-64

graphics. The TAB(X) function will be new to you. It has a similar function to the tabulator on a typewriter. The argument of the TAB(X) function determines where a character is printed on the screen. NOTE that the argument is always counted from far left. Hence TAB(26) means that the character following it, is placed on the 26-th location, reckoned from the left side of the screen.

QUESTION: what is the function of the reversed "S" after PRINT in line 41Ø? Consult your User Manual to find out.

```
4ØØ REM *** PRINT EVALUATION ***
4Ø5 :
41Ø PRINT"■"TAB(9)"NUMBER OF DICE ROLLS: ";RC
42Ø PRINTTAB(9)"=========================="
43Ø PRINT:PRINT
44Ø PRINTTAB(9)"SIDES   |  SCORE   |(THREE)
445 PRINTTAB(9)" ———————|——————————|———————————
45Ø PRINTTAB(9)"    1   |";SC(1);TAB(26)"|";TH(1)
455 PRINTTAB(9)"        |          |
46Ø PRINTTAB(9)"    2   |";SC(2);TAB(26)"|";TH(2)
465 PRINTTAB(9)"        |          |
47Ø PRINTTAB(9)"    3   |";SC(3);TAB(26)"|";TH(3)
475 PRINTTAB(9)"        |          |
48Ø PRINTTAB(9)"    4   |";SC(4);TAB(26)"|";TH(4)
485 PRINTTAB(9)"        |          |
486 PRINTTAB(9)"    5   |";SC(5);TAB(26)"|";TH(5)
487 PRINTTAB(9)"        |          |
488 PRINTTAB(9)"    6   |";SC(6);TAB(26)"|";TH(6)
490 RETURN
495 :
```

7.10 Final considerations

The another go routine may be identical to the ones developed in the previous chapters but as there is no database to assess, what BASIC statement can be omitted?

Are there any specific errors to watch for? Only the bugs caused by the usual typing errors, and you are now able to rectify them yourself without any indication from our side. Logic errors are not likely to occur but will happen when, for example:

- in the evaluation routine the AND-AND condition is not followed by the complete sequence Y = DI(1): TH(Y) = TH(Y) + 1, but the latter part is placed on the following line.

- in the dice roller the roll count is placed within the loop, instead of outside as it is now.

Suggestions for conversion to a game: you may delete the present display routine altogether, and also the evaluation routine. Maintain the initialisation routine and insert an input routine that prompts the user to enter three guess numbers.

The program will then generate its own numbers and attempt to match them with the ones you just entered. If there is one match, you get a point, two matches 10 points, and three matches 100. Hence, you have to create a routine that not only matches the input against the randomly generated numbers, but also counts them. Of course, it is not hard to think of some other variations.

To conclude this chapter, here is a complete routine that you may address with a GOSUB from line 205. When you add this subroutine, your program will be enriched with a 'funny effect'. Try to find out for yourself how it works.

```
2000 REM *** THE DICES ARE ROLLING ***
2005 :
2010 X$=" R DICES ARE ROLLING ■ "
2020 Y$ ="DICES ARE ROLLING"
2025 P$=CHR$(17):PQ$=P$
2030 FOR R = 1 TO 20
2034 PQ$=PQ$+P$
2035 PRINTPQ$
2040 PRINTTAB(Q)X$
2050 PRINTTAB(Q)Y$
2055 Q=Q+1:PRINT" □ "
2060 NEXT R
2070 Q=0
2075 PRINT " □ "
2080 RETURN
2095 :
2100 REM " R ■ " STAND FOR REVERSED FIELD ON AND OFF
```

APPENDIX I
Answers to some questions raised in the chapters

3.7

A string variable of the type A$, B$, AA$, XY$, NAME$, etc. The dollar sign indicates a string variable.

3.18

You must avoid, and therefore delete, the lines where the same operation is unnecessarily repeated.

3.24

If the flag does not equal 1, which is effectively the case when the flag equals zero, then the test proves to be 'false' anyway. Hence, the program flow 'drops through' and will stop in the END line.

3.26

The handling of GET is also explained in chapter 4, paragraph 4.12

5.4

Take for example X = 12.3456789. Working this out with X = INT(X * 1ØØ + .5)/1ØØ, results in:

12.3456789 * 1ØØ + Ø.5 = 1235.067890
Take the Integer: 1235.ØØØØØØ
Divide by 1ØØ : 12.35

5.9

With $(1 + I)^N - 1$ on the right side of the division slash, the computer will first divide by $(1 + I)^N$ and then subtract 1. With $((1 + I)^N - 1)$ it will divide the result obtained with the right hand part of the formula placed within the outer brackets: it first calculates (1+I), raises the result to the N-th power, then subtracts 1, and only after this sequence of operation has been completed, will it perform the division.

5.11

Without 'Clear Screen' the screen starts scrolling and the screen lay-out becomes cluttered. I=I/1ØØ is necessary, because entering a percentage means a figure divided by 1ØØ.

7.10

RESTORE can be omitted, because no data need be RESTORED.

APPENDIX II
Clues to solution of programming problems

2.10

(line number) INPUT "TYPE IN NR OF MONTH: "; NR
or
(line number) PRINT "TYPE IN NR OF MONTH: ";:
(line number) INPUT NR

2.12

There are twelve subroutines and their addresses (in the form of line numbers) differ each 40

3.6

1000 REM DATA STATEMENTS FOLLOW HERE
1010 DATA "JANUARY 10 JOAN DAVIDSON"

3.7

320 READ (follows string variable)
350 PRINT (follows string variable)
370 GOTO 320

3.9

9999 DATA "place here end of file name"

3.25

Line 41Ø — a blank line now and then makes the screen lay-out more readable

```
42Ø  INPUT "follows screen prompt"; FG$
43Ø  IF FG$ = "Y" THEN F=1: GOTO 45Ø
44Ø  IF FG$ = "N" THEN etc
445  GOTO 42Ø
45Ø  RETURN
```

4.7

```
21Ø  INPUT "SURNAME"; SN$
22Ø  INPUT "INITIAL"; FI$
23Ø  NM$ = SN$ + "/" + FI$
```

4.9

```
31Ø  LG = LEN (NM$)
32Ø  READ stringvariable (for example DA$)
33Ø  LG$ = LEFT$ (DA$, LG)
34Ø  IF LG$ = NM$ THEN PRINT DA$
```

4.12

```
41Ø  PRINT: REM BLANK LINE
42Ø  PRINT "ANOTHER GO? PRESS Y OR N"
43Ø  GET X$
44Ø  IF X$ = "" THEN 43Ø
```

(or: 43Ø GET X$: IF X$ = "" THEN 43Ø — in this case line 44Ø becomes obsolete)

```
47Ø  GOTO 43Ø
```

5.7

All program lines can be found in the text of this paragraph.

5.8

Consider in particular line 43Ø:
43Ø PRINT "ANNUAL INTEREST RATE (%):"; (I * 1ØØ)
What happens if you do not type (I * 1ØØ), but simply I?

6.6

12Ø DIM SC$(15)

6.7

5ØØ INPUT CT
51Ø IF CT<1 OR CT>6 THEN (clear screen): GOTO 41Ø
52Ø RETURN

or: 5ØØ GET CT: IF CT=Ø THEN 5ØØ

6.8

61Ø I = Ø
62Ø READ DA$
63Ø IF VAL(LEFT$(DA$,1)) = CT THEN GOSUB 2ØØØ

6.9

2ØØØ REM ARRAY FILLER
2Ø1Ø I = I + 1
2Ø2Ø LW = LEN (DA$) – 2
2Ø3Ø SC$(I) = RIGHT$(DA$,LW)
2Ø4Ø RETURN

6.10

8ØØ REM RANDOM GENERATOR
81Ø RN = INT (RND(1) * 1Ø) + 1
82Ø WR$ = SC$(RN)
83Ø LN = LEN(WR$)
84Ø LL$= etc, etc

6.11

```
1Ø8Ø  INPUT GW$
1Ø9Ø  IF GW$ = WR$ THEN PRINT: PRINT "WELL DONE etc . . .":
      GOTO 115Ø
```

7.6

```
21Ø  FOR I = 1 TO 3
22Ø  DI(I) = random generator for numbers 1 to 6
23Ø  ON DI(I) GOSUB etc
24Ø  NEXT I
25Ø  RC = RC + 1
```

7.8

```
41Ø  This line can be found in the text.
```

APPENDIX III
Screen Dump Routine for Commodore 64 Printer Combination

The program below is a so-called 'screen-dump routine' and it will be very useful if you have a line printer connected to your C-64. It may be incorporated as a subroutine into any program that displays output you deem important enough to obtain 'hard copy' from, i.e. a printout on paper.

When you activate this subroutine, for example by pressing the question mark key, it will 'dump' the contents of the screen. One way of activating it is to set aside a conditional test in combination with a GET . . statement. For example:

```
1ØØØ GET X$ : IF X$ = "" THEN 1ØØØ
1Ø1Ø IF X$ = "?" THEN GOSUB 6ØØØØ
1Ø2Ø GOTO 1ØØØ
```

IMPORTANT

1. This screen dump routine does not transmit Commodore's graphic characters.
2. Make sure that the rest of the program you incorporate this screen dump routine in, does not employ the same variables.

```
6ØØØØ REM *** SCREEN DUMP ROUTINE ***
6ØØ1Ø :
6ØØ2Ø OPEN 4,4: REM — OPEN LINE TO PRINTER
6ØØ3Ø FOR M = 1Ø24 TO 2Ø23 STEP 4Ø
6ØØ4Ø FOR SC = M TO M+39
6ØØ5Ø GOSUB 6Ø14Ø
6ØØ6Ø PRINT#4,CHR$(PP);:
```

```
60070  NEXT SC
60080  PRINT#4,""
60090  NEXT M
60100  PRINT#4:CLOSE4
60110  RETURN: REM END OF ROUTINE
60120  :
60130  :
60140  PP = PEEK(SC)
60150  IF PP< 32 THEN PP=PP+64: RETURN
60160  IF PP< 64 THEN                RETURN
60170  IF PP< 96 THEN PP=PP+32: RETURN
60180  IF PP<128 THEN PP=PP+64: RETURN
60190  PP = PP -128
60200  GOTO 60150
```

INDEX

COMMODORE 64 SOFTWARE PROJECTS

Please fill out this page and return it promptly in order that we may keep you informed of new software and special offers that arise. Simply fill in and indicate the correct address on the reverse side.

Name .....

Address .....

..... Code

What product did you purchase? .....

Which computer do you own? .....

Where did you learn of this product?

☐ Magazine. If so, which one?

☐ Through a friend

☐ Saw it in a Retail Shop

☐ Other. Please specify .....

Which magazines do you purchase?

Regularly:

Occasionally: .....

What Age are you?

☐ 10-15 ☐ 16-19 ☐ 20-24 ☐ Over 25

We are continually writing new material and would appreciate receiving your comments on our product.

How would you rate this software?

☐ Excellent ☐ Value for money
☐ Good ☐ Priced right
☐ Poor ☐ Overpriced

Please tell us what software you would like to see produced for your COMMODORE.

_____

_____

_____

PUT THIS IN A STAMPED ENVELOPE AND SEND TO:

In the United States of America return page to:
Melbourne House Software Inc., 347 Reedwood Drive,
Nashville TN 37217.

In the United Kingdom return page to:
Melbourne House (Publishers) Ltd., Melbourne House, Church Yard,
Tring, Hertfordshire, HP23 5LU

In Australia & New Zealand return page to:
Melbourne House (Australia) Pty. Ltd., 70 Park Street,
South Melbourne, Victoria, 3205.

NOTES

NOTES

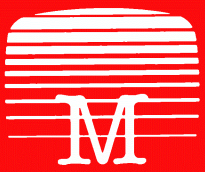
NOTES

NOTES

NOTES

NOTES

C64



Melbourne House

This is a very special book for anyone who wants to create useful programs on their Commodore and learn more about programming.

SOFTWARE PROJECTS will help you develop useful programs. You will not be given completed programs to type in! SOFTWARE PROJECTS gives you program ideas, discusses the overall set-up and an outline of the various procedures that are required to make up the program.

This book gives you help all the way. At each stage you can test what you have written, and in the unlikely event that you cannot quickly find a good solution to any problem, SOFTWARE PROJECTS offers you clues and possible answers.

You can create six useful programs by using this book, ranging from information management, business programs and games.

Special Features

- If you want to write useful BASIC programs, even if you have not written programs before, this book will show you how you can do so by yourself, giving you assistance at every step.
- SOFTWARE PROJECTS will improve your programming skills, show you how to organise your programs most efficiently and enable you to write and design your own program

Melbourne House Publishers

ISBN 0-86161-146-2

9 780861 611461