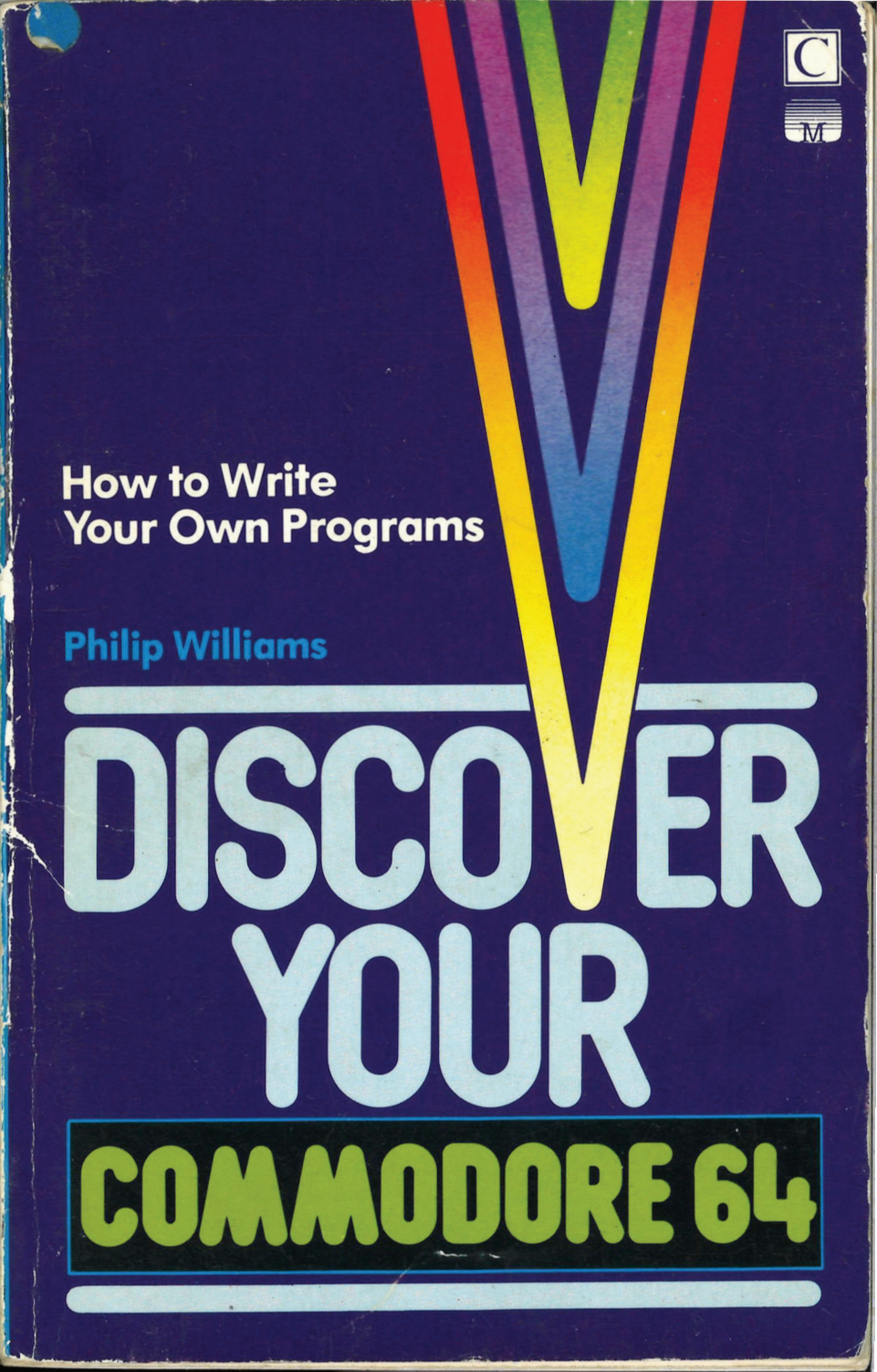

How to Write
Your Own Programs
Philip Williams
DISCOVER
YOUR
COMMODORE 64

DISCOVER YOUR
COMMODORE 64

DISCOVER YOUR
COMMODORE 64

CENTURY PUBLISHING CO. LTD.
WITH
MELBOURNE HOUSE (PUBLISHERS) LTD.

Published in the United Kingdom by:
Century Publishing Co. Ltd
Portland House,
12-13 Greek Street,
London W1V 5LE
in conjunction with
Melbourne House (Publishers) Ltd.,
Church Yard,
Tring, Hertfordshire HP23 5LU
ISBN 0 7126 0422 7

This book is published with acknowledgement of the contributions of Rudolf Smit.

Reproduced, printed and bound in Great Britain by
Hazell Watson & Viney Limited,
Member of the BPCC Group,
Aylesbury, Bucks

CONTENTS

INTRODUCTION :
Why this book

DON'T PANIC

This introduction tells you about the purpose of this book and how to use it.

I assume that you have unpacked your computer, plugged in all the cables, and even run some games on it. Perhaps you tried some 'real' programs as well. But now you feel at a loss. Although you understand the first few pages of your manual, the rest seems hard to follow. I know this feeling of panic — far back in the stone age of microcomputing I bought one of the very first micros, but then I had to do it all on my own with the most terrible manual anyone could think of. But, believe me, persistence won out after all!

IT IS NOT HARD

Therefore, do not feel ashamed because you do not know (as yet) how to work with your computer, let alone writing THE program you had in mind for such a long time, and which you were made to believe was so easy to write. Just as with any skill, programming skills are not acquired overnight. It takes time, and again, persistence, but it is not hard. It also takes a good guide, which is what this book endeavours to be.

FOR WHOM?

This book has been written for you, the novice to programming. It avoids the use of jargon for its own sake and tries to explain the art of computer programming in plain English.

In the second instance this book will also be helpful if you already HAVE acquired some programming skills,

i.e. if you have written programs and made them work as well. Making a program WORK does not necessarily mean that it is a GOOD program. In fact, it may be a terribly sloppy program, for reasons that will be outlined later. Most of this book deals with programming techniques which will give you a head start with any program you attempt and a professional finish when it is completed.

THE OTHER SIDE OF THE COIN

Perhaps more so than in other fields, home computing is something you have to do yourself. The big advantage of self-learning lies in the fun, creativity, and the tremendous satisfaction you can obtain from it. But the other side of the coin is that unless you learn properly, you may deprive yourself of all kinds of invaluable techniques which have been developed over the years by people whom we now deem masters of the art.

THE POPULAR LANGUAGE

This general observation definitely applies in the field of computer programming. Specifically so where the most popular computer language of the present time, BASIC, is involved — which is the language of your computer. BASIC programmers who learnt it all by themselves often have to UNLEARN — much to their surprise — a lot of undesirable programming habits before they can start learning and applying other computer languages.

JUMP AND SWIM?

Too often, the way people who try to learn programming are advised to 'jump into the water and swim'. This is not the aim of this book. In order to 'unlearn' already acquired bad programming habits and in order to acquire techniques which are generally recognised as proper programming skills, you will need to go back to square one.

A WORD OF ADVICE

Hence the advice is — follow this book from beginning to end. You should ideally start at the first chapter and work through the book, unless the instructions indicate otherwise.

The chapters are arranged in sequence. There are exercises in some chapters. Make a real effort to attempt them to gain the maximum benefit from this book.

IMPORTANT

This book is not meant to replace your Commodore-64 User Manual. It does not tell you how to get your C-64 going. Carefully study chapters 1 and 2 of your manual. In particular chapter 2, which will teach you how to use the keyboard, manipulation of the cursor movements, and also how to save and load programs. This is essential reading.

CHAPTER 1

Hurdles to good programming

This chapter outlines the idea of what is considered bad programming and what is generally recognised as good programming. But before we can discuss this matter in detail we had better ask ourselves the question 'what is computer programming anyway?'.

This is not a silly question — there are many people who have an incorrect idea of what computer programming is all about.

PROGRAM MEANS LIST

First, let me make it clear that in the computer industry the American word 'program' is preferred to the English word 'programme'. Second, the notion of program can convey many ideas. We talk about radio and television programs, the program of a day's outing, a political program, the program of a night at the concert hall, etc.

The differences between all these types of programs are obvious, but they all have one thing in common. The essence of a program is a LIST of words conveying messages, meanings; but in the field of computers, A PROGRAM IS A LIST OF INSTRUCTIONS.

AN EXAMPLE

To make this clear to you let me bring in an example from everyday life.

Suppose that you have never driven a car in your life and you want to learn. The logical step is to contact an instructor.

THE CORRECT SEQUENCE

Will the proper understanding of all that enable you to drive a car? Certainly not.

The instructor will then tell you the correct sequence of all the physical actions which will enable you to drive the car. For example, there is only one way to get the car into action, i.e. starting the engine by turning the ignition key while giving some gas. The next steps will be to engage the clutch, switch into gear, give some gas, let the clutch come up, and so on.

FOLLOW THE RULES

It can't be done in a different way. A fixed set of rules have to be followed and a particular range of consecutive instructions have to be carried out before you get the car moving from A to B.

The same principles apply when, instead of a car, you want a computer to function properly. After all, just as with a car, a computer is nothing but a machine which can't do things on its own. It needs a series of instructions in a very precise sequence, before it can perform the task the user wants it to do.

That is the crux of the matter — these instructions, before being fed into the computer's memory, have to be written very carefully, i.e. in EXACT wording, and in exact sequence. If you do not follow these rules, there is a good chance that the computer 'does not understand' the instructions (when they are not written correctly) or the computer goes berserk when the instructions are written correctly but in the wrong sequence. In the first case we talk about SYNTAX ERRORS and in the second case we talk about LOGIC ERRORS.

A SIMPLE EXAMPLE

To illustrate what I just said, let me present a very simple program. Type in this program exactly as it is written here:

```
10 A = B + 1
20 B = A
30 PRINT A
40 GOTO 10
```

NOTE: I hope that you have already typed in some simple programs from your computer manual, so that you know how to 'enter' a program. If not, here is a short explanation.

Switch on your machine (and naturally your TV set or monitor, connected to the computer). Then type:
10 (space)A(space)=(space)B(space)
+(space)1 and then hit the RETURN key. The program line (because that is what it is) is now firmly locked into the computer's memory. Now do the same with lines 20, 30 and 40.

TO RUN A PROGRAM

The next thing to do is 'running' the program. You do this by typing RUN and then hit the RETURN key.

What you see next is how, with dazzling speed, numbers from 1 onwards are scrolling over the screen. Before you know it this program has made a thousand counts (that is what it is doing — counting) and unless you stop it, by hitting the RUN-STOP key or by switching off the computer, it will go on seemingly forever.

TO STOP A PROGRAM

Well, you have seen what the program does so stop it, by hitting the RUN-STOP key. By the way, you will then see a message on the screen: BREAK IN (followed by one of the line numbers) and READY. If you want the program to continue, type CONT and hit RETURN.

AN ERROR

Now let us introduce an error deliberately.

Type exactly:

```
30 PRIN A
(hit the RETURN key)
```

And RUN the program again. The computer will now respond with the message ?SYNTAX-ERROR IN 30. Why? Simply because you should have typed PRINT instead of PRIN. A human may still recognize the word PRIN, that is to understand that the command PRINT is meant by it. The computer on the other hand is not that clever and simply stops the execution of the program. The programmer must rectify this syntax error before the computer will ever accept this program.

STATEMENT ERROR

Now let us do another trick. Type exactly 40 GOTO 01. And RUN the program again. No way! The computer stops execution immediately and throws this message on the screen: ?UNDEF'D STATEMENT ERROR IN 40. This is a typical LOGIC ERROR. The program logic, the list of instructions, is wrong because the computer is directed to a program line (01 or as the computer sees it, 1) which is not there! Line 1 has not been defined which is why the computer talks about UNDEFINED STATEMENT ERROR.

WRONG ANSWERS

This is a very clear logic error. But there are also logic errors which are not so clearly visible. That is the case where the program appears to perform correctly but comes up with the wrong answers to the problem it is supposed to solve.

Back to our example. Now type in:

```
40 GOTO 20
```

and RUN the program.

What you now see is a series of 1's scrolling over the screen at dazzling speed. Well, quite obviously, this is not what the program is supposed to do.

The logic error lies in the fact that the programmer has directed the computer to the wrong program line. As a consequence it is only repeating a statement (B = A) instead of performing the task of adding up (in line 10 where A = B + 1).

INFINITE OR ENDLESS LOOP

One of the nastiest logic errors a programmer can make is the 'infinite loop'. Just try this — type NEW and hit RETURN first, to erase the previous program from the computer's memory:

```
10 GOTO 10
```

The syntax of this program is perfectly okay — every letter and digit have been typed in correctly. But when you RUN it, it will perform a totally senseless task: running around in circles forever.

The program line 10 GOTO 10 is constantly directing the computer to line 10. It is an endless loop from which it can't escape, unless you hit the RUN-STOP key or switch off the computer.

A SIMPLE EXAMPLE

The program at the beginning of this paragraph is in fact also an endless loop, but at least it is still performing a task within that endless loop.

Alright, you have now an idea of what a computer program is all about. And naturally, we will expand on it considerably in the forthcoming chapters. But let us now concern ourselves with the question about good and poor programming techniques.

GOOD AND POOR PROGRAMMING

What is good and what is poor programming? Instead of theorizing at length about all the possible answers, I think that you will initially be best served by studying carefully the two examples printed on the next two pages.

NOTE: before we start to discuss them more thoroughly, I want you to realise that for the sake of clarity both programs are exaggerated in appearance and that they solve the problem for which they have been written in a rather oversimplified manner.

Both programs will produce the same correct result when the same data is entered into them. But if that is so, then your first impression may be that both programs are good because they produce the desired result. You may even think that the shorter program is the better one because it is shorter.

The crux of the matter is that at present a correct result is considered a less important criterion than the way the program has been designed and presented. For that matter a short program is not always the best designed program!

PROGRAM A

```
5 PRINT"J"
7 X=60
10 INPUTH:INPUTM:INPUTS
20 S=S/X
30 M=M+S
40 M=M/X
50 H=H+M:PRINTH
60 END
```

EXAMPLES

Now have a closer look at both examples. You do not have to be an experienced programmer to discover that example A (although considerably smaller) is a lot harder to read than the much larger example B.

Now suppose that you ARE an experienced programmer. This does not at all guarantee that you can figure out at once what program A is supposed to do. On the contrary, if you are not familiar with the problem this program can solve, it will remain a puzzle to you.

Program B on the other hand, tells you at once for what purpose it has been written. The layout is nice, almost every line has been commented (by so-called REM(ark) statements), and lines have been left open to improve readability. Program B truly has a professional touch.

Program A, although it does perfectly the job it is designed for, is an example of poor programming. (Of course, it will not be necessary in every instance to go to the extremes of program A.)

PROGRAM B

```
100 REM ***************************
110 REM *                         *
120 REM * HOUR & DEGREE CONVERTER *
130 REM *                         *
140 REM ***************************
150 :
160 REM THIS PROGRAM CONVERTS HOURS MINUTES AND SECONDS INTO DECIMAL HOURS
170 REM OR DEGREES MINUTES AND SECONDS INTO DECIMAL DEGREES
180 :
190 REM AUTHOR        EDWARD  BARTON
200 REM DATE WRITTEN  01 - 12 - 1983
210 :
220 PRINT"□"                    :REM CLEAR SCREEN
230 :
240 REM INPUT SECTION
250 :
260 INPUT"HOURS  ";HR           :REM STORE HOURS IN HR
270 INPUT"MINUTES";MN           :REM STORE MINUTES IN MN
280 INPUT"SECONDS";SC           :REM STORE SECONDS IN SC
290 :
300 REM CALCULATION SECTION
310 :
320 SC = SC/60                  :REM MAKE SECONDS DECIMAL
330 MN = MN + SC                :REM ADD DECIMAL SECONDS TO MINUTES
340 MN = MN/60                  :REM MAKE MINUTES DECIMAL
350 HR = HR + MN                :REM ADD DECIMAL MINUTES TO HOURS
360 :
370 REM PRINT FINAL RESULT (PLACED IN HR) AFTER PRINTING ONE BLANK LINE
380 REM REVERSED SYMBOL Q PRINTS ONE BLANK LINE
390 :
400 PRINT"■CONVERTED TO DECIMALS :";HR;" HRS"
410 :
420 END
READY.
```

DOCUMENTATION IS VERY IMPORTANT

Let us discuss another example, which on the surface does not have anything to do with programming, but will give an extra idea as to what we are aiming for.

Ask an experienced car machanic to repair a car he has never seen before. No doubt he will understand the basics of the car, but when it comes to details he will be at a loss. Therefore what he will do next is to call on the manufacturer or distributor of the car to obtain all the technical documentation he needs.

But that won't be enough. In order to work as efficiently as possible this documentation must be complete, concise and easy to follow. If the mechanic has to write question marks all over the place, if he finds that certain pages or even entire sections are missing, or that some are outdated, then this documentation can be qualified as sloppy. The result is a frustrated mechanic.

Back to programming. Now imagine that you have bought listings of programs you like to experiment with.

I assure you that you will develop strong feelings of despair when you are presented with an immense

word puzzle, instead of a well documented, easy to follow program.

Now return to examples A and B. At this moment in time you may understand the purpose of program A. But when you again see program A after, say, a year's time, you will not understand it. On the other hand, when you see program B, say twenty years later, you will grasp, virtually at once, what the program is all about — thanks to its documentation, and the way it has been presented. It is a well structured program.

RULES OF GOOD PROGRAMMING

Good documentation is only helpful when the program is properly designed. Therefore it is imperative to follow at least some rules when you design a program.

(1) the programming job should be well prepared. At least you shall have to specify as precisely as possible what you hope the program will do for you;

(2) the program should be neatly structured; that is, it will be built up in blocks and each block performs a specific task;

(3) the program should have a comment for almost every line;

(4) in general terms the program must be easy to follow. Thanks to this and its documentation throughout, it can be easily understood and modified or updated if need be.

It is hard truth that many of the existing programs in BASIC are lacking virtually all of the qualities explained. Therefore most of them can be described as examples of sloppy programming, although they may do a good job anyway.

What I have just defined as rules which make a program a good program, are some of the rules pertaining to the art of 'structured programming'. We will go into more detail about this later on, but be sure that the major benefit to you in following these rules is that programming is easier this way!

STRUCTURED PROGRAMMING — WHAT IS IT?

It is perhaps more straightforward to define the opposite of structured programming. It is the sort of programming which can be described as the consequence of the trial-and-error approach. Which, in turn, is based on the general assumption that programming is: seat yourself in front of your keyboard and video monitor, start typing and just see what happens. Every time you make an error the machine will duly report it and then you will attempt to rectify it.

And so, line by line, you will write your program, changing and interfering at any time you wish. It will produce programs which are effective, that is, they do the job. But are such programs efficient? In most cases no!

Quite often these are 'jumping-all-over-the-place' programs. Only the authors of these programs understand their creations . . . initially. For experience tells us that after some time they lose track in their own programs.

METHOD AND SYSTEM

Structured programming, provided it is done in accordance with the rules, is the total opposite of the unsystematical trial-and-error approach. Structured programming is indeed about structure, the essence of which is method and system, which leads to clarity and the ability to recognise. Aside from qualities which I already pointed out — documentation, design in program blocks, easy-to-follow, and so on — the main characteristic of structured programming is the so-called TOP-DOWN approach.

TOP-DOWN APPROACH

What does that mean? Essentially the principle is quite simple. A program starts at the top and ends at the bottom. But there is far more to it of course, otherwise it would not be such an important topic within the entire computer world.

The principles of the top-down approach are to be understood to their fullest extent only when you have worked out a large program according to its rules. Which is why I do not dwell too long on this subject — the next chapters will explain it fully.

But let me emphasise that the idea of top-down is maintained throughout every part of the program. As explained previously, a good program is well structured in that it is built up in separate blocks and each block performs a specific task.

SUBROUTINES

Such a block may be considered as a small, separate program, more appropriately called a sub-program, or as BASIC programmers prefer it: a SUBROUTINE.

In agreement with the principles of the top-down approach a subroutine begins at the top and ends at the bottom. Whereas, and this is most important, the top-end marks the ONLY entry point of the data which are to be processed within that subroutine, and the bottom-end marks the ONLY point to exit the subroutine.

ENTRY & EXIT

The general rule in structured programming is that a subroutine may not have more than one entry-point and not more than one exit-point. To clarify this a little more, compare it with a factory: there is usually one area where the raw materials are received, and there is usually another area where the finished product leaves the factory.

Now back to a subroutine: the entry-point receives the data to be processed, and the exit-point delivers the correct results. In computer jargon: the entry-point receives the INPUT, the exit-point RETURNS the OUTPUT.

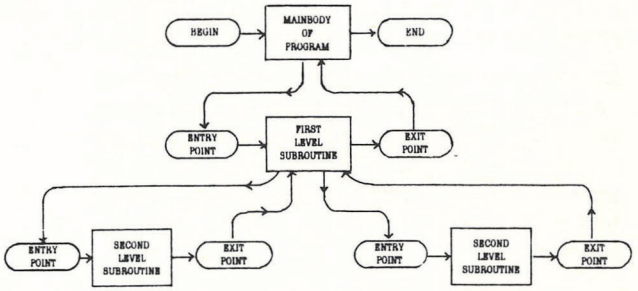
Later on we will go into far more detail about subroutines, but let me conclude by conveying the general consensus among programmers that subroutines should be as short as possible — preferably not longer than one page (A4 size) if printed out. If the programmer feels the need to refer to lower-level subroutines within a subroutine he is

perfectly entitled to do so, particularly when he wants to adhere to the rule that a subroutine performs one task only.

LOWER-LEVEL SUBROUTINES

Inadvertently I already used the notion 'lower-level subroutine', without explaining it. We will discuss that topic now. Another rule pertaining to the principle of top-down approach, is that subroutines are placed within the program in the order of their level.

To illustrate this, look at the diagram printed below. It depicts the top-down approach to its fullest extent.



You see here, at the top, the main body of a program (programmers prefer to call it the MAINLINE) which is doing the actual job, because it is there where it begins and also where it ends. However, it utilizes a subroutine, to which it assigns a part of the processing.

As you see, a 'flowline' (a line along which the program flows) branches out from the main body and enters the lower-level subroutine — here called the first level subroutine — and at the exit-point the flowline returns to the main body. The first level subroutine, however, utilises two subroutines, which consequently are called subroutines of the second level. As you see, these second-level subroutines are placed below the first-level subroutine.

In real programming we can't place subroutines next to each other, as shown in the diagram. The way a program with a mainline, one first-level subroutine, and two second-level subroutines, is built up, is simply:

(1) Mainline-routine

(2) First Level Subroutine

(3) Second Level Subroutine

(4) Second Level Subroutine

EXAMPLE

To give a live example, we examine again the example of the HOUR & DEGREE CONVERTER (program B) but now written according to the rules of the top-down approach. See program C.

NOTE: If some readers find that this is truly exaggerated example of structured programming, then they are right. This has been done this way for the sake of clarity.

```
100 REM ****************************
110 REM *                          *
120 REM * HOUR & DEGREE CONVERTER *
130 REM *                          *
140 REM ****************************
150 :
160 REM THIS PROGRAM CONVERTS HOURS MINUTES AND SECONDS INTO DECIMAL HOURS
170 REM OR DEGREES MINUTES AND SECONDS INTO DECIMAL DEGREES
180 :
190 REM AUTHOR        EDWARD  BARTON
200 REM DATE WRITTEN  02 - 12 - 1983
210 :
220 PRINT"⍰"                    :REM CLEAR SCREEN
230 :
240 REM-------------------------------------------------------------------------
250 REM MAINBODY OF PROGRAM
260 GOSUB 320                   :REM INPUT OF DATA SUBROUTINE
270 GOSUB 420                   :REM CALCULATION SUBROUTINE
280 GOSUB 510                   :REM PRINT RESULT SUBROUTINE
290 END                         :REM END OF MAINBODY OF PROGRAM
300 REM-------------------------------------------------------------------------
310 :
320 REM INPUT SUBROUTINE
330 :
340 INPUT"HOURS  ";HR          :REM STORE HOURS IN HR
350 INPUT"MINUTES";MN          :REM STORE MINUTES IN MN
360 INPUT"SECONDS";SC          :REM STORE SECONDS IN SC
370 RETURN                      :REM RETURN TO MAINBODY OF PROGRAM
380 REM-------------------------------------------------------------------------
390 :
400 REM CALCULATION SUBROUTINE
410 :
420 SC = SC/60                  :REM MAKE SECONDS DECIMAL
430 MN = MN + SC                :REM ADD DECIMAL SECONDS TO MINUTES
440 MN = MN/60                  :REM MAKE MINUTES DECIMAL
450 HR = HR + MN                :REM ADD DECIMAL MINUTES TO HOURS
460 RETURN                      :REM RETURN TO MAINBODY OF PROGRAM
470 :
480 REM-------------------------------------------------------------------------
490 REM PRINT RESULT SUBROUTINE
500 :
510 PRINT                       :REM PRINT ONE BLANK LINE
520 PRINT"CONVERTED TO DECIMALS :";HR;" HRS"
530 RETURN                      :REM RETURN TO MAINBODY OF PROGRAM
540 REM-------------------------------------------------------------------------
550 REM END OF PROGRAM
```

PROS & CONS

Honesty compels me to tell you about the heated debate which has been going on about the alleged disadvantages of structured programming, particularly in relation to programming in BASIC. The

most often heard argument against structured programming is that 'it kills creativity'. The trial-and-error approach certainly has some charming aspects, but to our firm conviction these do not outweigh its disadvantages which have been exposed extensively in the previous paragraphs.

SLOW EXECUTION

Another argument against structured programming is of a more technical nature. Comment lines consume memory space, and slow down the execution of the program. This argument is a leftover of the early days when microcomputers were indeed very limited in memory space. To make a computer program as efficient as possible, so that it would fit into that very limited memory space, everything considered superfluous was left or thrown out. The most superfluous thing was a REM(ark) line. Hence the habit of writing programs without any comment line.

In those days programmers were indeed very concerned about the speed at which program executed. If it could be improved by a few tenths of a second, one would go through quite some trouble to achieve that goal. For example, it was soon realised that in BASIC a subroutine is searched for from the very first line in the program onwards. Which is why lots of programmers placed subroutines as high as possible in the program to obtain some extra tenths of seconds speed, whereas they started the program somewhere at the bottom! So you can see how undesirable programming habits came into being.

YOU NEVER HAD IT SO GOOD

The days of limited memory space are definitely over. Your Commodore 64 has, when it is powered on, over ten times more memory available than some of those early computers. You never had it so good!

I can assure you that you truly have to write a mighty long program to fill all that space. Which is why comment lines are not superfluous any longer! It is now also recognised that it does not really matter where a subroutine is placed as far as the speed of program execution is concerned.

CHAPTER 2
Tools of BASIC

ACQUIRING THE TASTE OF PROGRAMMING

My father taught me carpentry in a very simple way. The first time he allowed me in his workshop he showed me a few tools and said, 'This is a hammer, this is a saw, here are pliers, and there are nails. Now just watch what I am doing''. Only occasionally did he answer questions, and only sometimes did he stop to give some extra explanation. It was a no-nonsense approach, and I learnt a lot.

This is the way this book will introduce you to programming. In this chapter I will outline to you some of the very basic tools of the BASIC computer language.

A WORD ABOUT BASIC FIRST

The word BASIC does not imply that it is a very 'basic' language. In fact, it is an acronym for **B**eginners **A**ll-purpose **S**ymbolic **I**nstruction **C**ode, and it was developed by a team at Dartmouth College (California) during the sixties, purely to serve as an easy-to-learn computer language for beginners.

Probably much to the surprise of its inventors, BASIC has become the most popular computer language of the present time, and though it still has the image of a language for beginners, it has shed its beginners status long since and has grown into a fully professional language, available now on most microcomputers.

An unfavourable development however is the many different versions or 'dialects' which have become available. The BASIC of your computer is just one of many dialects. This implies that when you have mastered a particular version of BASIC you will have to do some additional learning and un-learning when you want to work with a different dialect of BASIC. However all BASICs have a standard set of instructions in common, hence a switch-over to a BASIC different from what you will have used is not difficult.

GENERAL RULES FOR BASIC

BASIC is a language, and as you know, every language has certain rules. We call that the SYNTAX of a language. Human languages show some flexibility in the application of their rules. Computer languages have virtually NO flexibility. They must be written strictly in accordance with the rules. Though many people attribute almost magical powers to computers, they are nothing but machines which can't think for themselves. Therefore, a computer will not recognise instructions which are not precisely written as the syntax of the language dictates.

For example, when you type INNPUT instead of INPUT, the computer will react with an error message on the video screen. A human will still recognise the word INNPUT and comprehend its meaning, but the computer recognises only the word INPUT (with one N).

NUMBERING OF LINES

A program in BASIC is written in lines containing the instructions which make the program work. Every line MUST be numbered, and a series of lines must be numbered in INCREASING order. It does not matter how you number these lines, as long as the previous line number has a lower value, and the next line number a higher value. For example the next sequence of BASIC instructions is numbered correctly:


```
0 B = 1
10 A = B + 1
99 PRINT A
111 GOTO 0
```

But you can also write:

```
1 B = 1
2 A = B + 1
3 PRINT A
4 GOTO 2
```

Or:

```
1000 B = 1
2000 A = B + 1
3000 PRINT A
4000 GOTO 2000
```

or any other sequence of line numbers as long as you abide by this rule of increasing numbers.

IMPORTANT: BASIC has made it easy for you to add, insert or delete program lines. For example, if you had just typed in the previous program, and you want to insert a line 1500, just type 1500 and then the desired instructions.

When you RUN the program, the computer will automatically place the new program line in its proper position. This applies for adding and inserting lines. When you want to delete (remove) a program line just type the number and hit the RETURN key. The line then disappears from the program. Removing, adding and altering lines comes under the heading of 'editing'. You will find that cursor movements are of great help. I refer here to Chapter 3, page 34, of your User Manual.

SPACING

Some computer languages have strict rules for spacing between instruction words. In most cases BASIC is flexible on this point. The computer will accept, for example:

```
1B=1
2A=B+1
3PRINTA
4GOTO2
```

This does not improve readability:

Justtrytoreadthisitisnotsoeasyisit? Your computer will accept

1 A = A + 1,

which is the other extreme. But IT WILL NOT ACCEPT SPACING BETWEEN THE LETTERS OF INSTRUCTION WORDS!

For example, it will accept READ as an instruction, but it will NOT accept R E A D.

When you do something which is against the rules (or syntax) of the BASIC computer language, your computer will report it to you at once, by putting an error message on the screen of your monitor or TV set. You can then correct this error (commonly called a SYNTAX ERROR) immediately, for example by typing the program line again. It is because of this special quality of BASIC (to report errors and enable the programmer to rectify them at once) that BASIC is called an INTERACTIVE language.

NUMBER WITH INTERVALS OF TEN

There is general agreement amongst programmers to number program lines with an interval of ten. The line sequence 1, 2, 3, 4, etc. will not allow you to insert extra program lines if the need arises (that need arises sooner than you think!). The sequence 10, 20, 30, 40, etc. on the other hand offers you ample room for inserting extra lines.

It is also good programming practice — certainly in the case of large programs — to begin with line number 100. When the program has been written completely, you may then add quite a few comment lines to the program in the area before line number 100.

HOW MANY INSTRUCTIONS ON ONE LINE?

Your computer will allow you to type as many as eighty 'characters' in a program line.

A 'character' is the general name for any letter, digit, dot, comma, graphic symbol, or whatever you have on your keyboard. Even an empty space is seen by the computer as a character!

For the sake of clarity I advise you to keep the length of a program line within 'reasonable proportions'. When you develop your own program, following the suggestions of this book, you will see what I mean by 'reasonable proportions'.

MAINTAIN CLARITY AND READABILITY

Remember the rules of structured programming. Clarity and readability are two of them. This boils down to three guidelines:

(1) Type spaces between the instructions. We have already discussed this in the paragraph about spacing.

(2) Use open program lines now and then. Commodore BASIC assists you greatly at this point. See this example:

```
10 INPUT A
15 :
20 PRINT A
```

The colon after 15 gives you an open line.

(3) Use comment lines which tell precisely what you are up to. Again BASIC assists you here with the so-called REM statement. REM stands for REMark. You can place them anywhere in a program as they do not interrupt the normal program flow. See this example:

```
10 INPUT A : REM INPUT TEST DATA
15:
20 PRINT A : REM DISPLAY TEST DATA
```

A COMPUTER NEEDS INFORMATION TO WORK WITH

Before we go into more detail about the most fundamental keywords of BASIC, we must discuss two words which you probably have heard about long before you even thought of getting a computer. These are the words 'input' and 'output'. They are 'jargon'

originating in the world of record players and amplifiers, but jargon or not, they have become so commonplace that we can consider them as part of everyday language. You may have heard people talking about someone else as a very energetic worker — 'he has a considerable output' — i.e. he manages to produce a lot within a short time.

INPUT & OUTPUT

When you set out to write a letter to someone, you will "put in" a lot of information. The result is the letter — your OUPUT — that you will send off by mail. I do not think that you will ever say 'I will input a lot of information into my letter to John', but this was just an example to establish the link with computers. You already came across them in the manual of the Commodore 64.

To elaborate: with 'input' you give the computer some information. When you 'feed' the computer some input, the computer will accept it and then work on it to produce a result which is called 'output' (the process of working on it is often summarised in the word 'throughput).

Input can be offered in more than one way. For example you can feed the computer with information recorded on tape or on a magnetic disk. But certainly, in your case, input of information can also be done by hand. You type it in via the keyboard when the program in the computer truly ASKS for input.

Output can also be produced in more than one way. Most commonly on the screen of the monitor or your TV set. Or in print (on paper) for which a printer is required.

For the time being we are only concerned about the typing in of information, which is 'manual input', and the results which you can view on the screen, which is 'screen output'.

GETTING STARTED

Now before we go any further, a soul-searching question — have you studied carefully and done all the exercises in Chapter 2 of your User Manual? If

not, I strongly suggest you do this first before we can proceed any further!

Chapter 2 of the User Manual is like the first driving lesson, where the instructor tells you such things as 'there is the brake, there is the clutch, here is the ignition-contact . . .' and so on. Chapter 2 is absolutely imperative — if you have not learnt as yet how to work with the cursor, i.e. moving it around on the screen, how to delete letters, digits, graphic symbols you just typed in, you will be nowhere as far as the rest of this book is concerned. Chapter 2 of the User Manual is indeed an excellent guide in learning about the basics of your machine.

SAVING YOUR PROGRAMS

Writing and testing a program is not a matter of a few hours. It often takes days, and even months when the job is large.

Consequently you must save your program intermittently during all its stages of development onto tape, or disk. After all, you can't keep your computer switched on forever. But when you switch off your computer, the program will be lost. Therefore, at the end of a day of programming you SAVE what you have developed so far onto tape, or if you have a disk-drive, onto disk. The next time you LOAD what you have developed so far from disk or tape, and you continue the job.

SAVEing and LOADing from cassette tape is very easy. Your Commodore-64 manual explains it clearly. Working with disks, however, is a different matter. Appendix A at the end of this book will provide you with clear instructions regarding 'disk handling'.

SOME COMMANDS IN BASIC

We start out with just a few instruction words, more commonly called commands. With these commands and a few other symbols you can already write simple programs. These instructions are also known as 'keywords'.

INPUT & PRINT

What they do: with the INPUT command you can feed the computer and its program with the required information. The PRINT command will display the result (output) on the screen, or in combination with other instructions, on the paper of a printer. How they do it:

Switch on your computer and TV or monitor. Type exactly:

10 INPUT A
(and hit the RETURN key)
20 PRINT A
(and hit the RETURN key)

RUN this tiny program (type RUN and hit the RETURN key).

You will see a question mark on the screen with a flashing cursor just after it. Now type 23 and hit the RETURN key. You will see the computer displaying another number 23 just below the one which you just 'inputted'. What did it do? This requires some elaboration.

LETTER CHARACTERS CAN STORE INFORMATION

What does the A mean in the program lines INPUT A and PRINT A?

To explain this let us take an example — a filing cabinet with many drawers. You will realise of course that a filing cabinet stores information. When the owner of such a filing cabinet is a methodical person, he will most certainly LABEL (giving a name) to its drawers, to make it easier to identify the information stored in that drawer. For example drawer A may contain household bills, drawer B may contain study results, drawer C a stamp collection, etc.

A computer has 'drawers' available in which it can store the information it needs to run a program properly. These 'drawers' MUST be identified by labels in the form of one or two letters, or a letter-digit combination. Examples: AS, AA, A1, AB, BX, X9, etc.

Now, when I write this program line:

10 INPUT A

I actually say: 'this line 10 of my program instructs the computer to store information, to be typed in via the keyboard, into a drawer which I identify with the letter A'.

And when I write this program line:

20 PRINT A

I actually say: 'this line 20 of my program instructs the computer to look into the drawer which I identified with the letter A, and report to me what it holds in store, via a message on the screen'.

Keep in mind that just as with a drawer of a filing cabinet, a 'drawer' in the computer will keep its label (its name), but it can also store different information. You can store household bills in this drawer, but later you may decide to throw that lot out, and replace it with your stamp collection. But whatever you do, the name of the drawer remains the same. The only thing that VARIES is the contents. This is why the computer world prefers to call these imaginary drawers VARIABLES.

VARIABLES ARE EXTREMELY IMPORTANT

These identifiers of 'information drawers', or more appropriately VARIABLES, form an extremely powerful programming tool. You will find hardly any program without them. They truly belong to the first building blocks of any program. When you have lots of information stored in variables, you can manipulate and shuffle them around as you wish. After all, a computer, yours included, is a manipulator of information. Take this example:

```
10 INPUT A
20 INPUT B
30 INPUT C
40 INPUT D
50 E = A + B + C + D
60 PRINT E
70 E = A * B * C * D
```

```
80 PRINT E
90 A = E
100 B = E + A
110 C = E + D
120 F = A + B + C
130 PRINT F
```

This example shows how the information (or 'data' as preferred by programmers) is manipulated in these program lines thanks to the use of variables. Moreover you will discover the true value of a program.

A program is a list of instructions, and every time you feed such a program with different data for INPUT, you naturally get a different output all the time. But that is precisely what a computer is expected to do.

For example, you can write a program that will convert for you any number of feet into metres, decimetres, centimetres and millimetres. A program that does such a job may look like this:

```
5 REM TYPE IN NR OF FEET
10 INPUT FT
20 MT = FT * 0.3048
30 PRINT "IN METRES:";MT
40 DM = MT * 10
50 CM = DM * 10
60 MM = CM * 10
70 PRINT"IN DECIMETRES:";DM
80 PRINT"IN CENTIMETRES:";CM
90 PRINT"IN MILLIMETRES:";MM
```

Just try out this program. You know the rules.

When you switch on the computer you can start typing in the program. When the computer is already switched on and there is a program in it, then type NEW and hit the RETURN key to erase that program.

Start a line with a line number and type the instructions. When the line is completed, hit the RETURN key.

Do not worry at present about the use of quotation marks, the colon (:) and the semi-colon (;). That all comes later. Just get this program running and you will see why variables are so important. Note also the variable NAMES: FT, MT, DM, CM and MM. They are all closely related to what they represent.

Every time you RUN this program you can type in any number of feet, for example 1, 20, 100 or 1000 feet, and this program will give you the exact answers.

+ AND − ARE 'ARITHMETIC OPERATORS'

The symbols +, −, × and = stand for the arithmetic functions we are all very familiar with in everyday life. As any computer is, in principle, a number-cruncher it is obvious that these four functions are part of every computer language.

For this purpose they have been called ARITHMETIC OPERATORS. After all, they operate, to a certain degree, the program flow in the computer.

When a computer program encounters the symbol + it will normally do the operation of adding up. The same applies, of course, for the other arithmetic symbols. However, as you have seen already, the computer uses somewhat different symbols for the multiplication and division functions.

In virtually every computer language the asterisk symbol (*) has replaced the × for multiplication. The computer can't see the difference between an × as a multiplication symbol or the X as the letter X. Therefore another symbol to represent the multiplication function became necessary.

The slash (/) as a symbol for division does not need extra explanation. It is quite common as the division symbol in the world of arithmetic and mathematics.

One arithmetic operator has not been considered so far. It is the vertical arrow (↑) which has the function to raise a number to a power. You all know how we write 'raise 2 to the 3rd power': 2^3. The standing arrow takes care of this. So if, for example, you want to know the result of 2 to the fourth power, you could write 2 * 2 * 2 * 2. But that is cumbersome. This will do the same job: 2 ↑ 4. Just try it on your computer:

PRINT 2 ↑ 4 and hit RETURN.

THE COMPUTER AS DECISION MAKER

The feature which truly makes a computer so powerful in the sense that it can do so many things, is the fact that it can test certain conditions and then make a decision. These tests are performed by the so-called RELATIONAL OPERATORS in combination with a few other commands.

Now what is a 'relational operator'? Quite simply a function which tests the relation between two pieces of information. The best known relational operator is the Equals sign (=). When I say A = B (A equals B) I state the relation between the factors A and B. They happen to be equal. I can also say A = B + C. The relation between both sides is then that the factor A equals the addition of the factors B and C.

However, we can also say A is NOT Equal to B, or A is greater than B, or A is smaller than B. In all cases we actually test the relationship between two factors, one on the left of the relational operator and the other one on the right.

IF the condition which is tested by those relational operators is met, or not met, the computer will make a decision either way.

For example, when the computer encounters something like:

```
200 A = B
210 IF A = 10 THEN 400
220 B = B + 1
```

it will test the relation A = 10 in line 210. When that relation is found to be 'true', that is when A does equal 10, then the computer will decide to direct the program flow to line 400. If not, it goes to line 220.

Much more about this fascinating feature in the next chapter, and let us conclude with the naming of the relational operators other than the equals sign:

< stands for **less than**
> stands for **greater than**
<= stands for **less than or equal to**
>= stands for **greater than or equal to**
<> stands for **not equal to**

THE FUNCTIONS OF . , : ; AND " "

The period (.) has only one function. It is the decimal point of a number with digits after the decimal point. Nothing else. If you try to use it to end a sentence, the computer will react with a SYNTAX ERROR.

The comma (,) separates PRINT outputs and DATA statements (more about the DATA statement in the last chapters of this book).

But if you, for example, write 100 INPUT A,B,C the computer will ask for input of three numbers, one after the other. On the other hand the program line 110 PRINT A, B, C will print on the screen the contents of A, B and C on the same horizontal screen line, though with some spaces in between.

The colon (:) separates commands. As you have already seen (in Chapter 1), program lines can be written this way:


```
10 INPUT A: INPUT B: INPUT C
```

This will have the same effect as:

```
10 INPUT A
20 INPUT B
30 INPUT C
```

In the first case the colon (:) is mandatory. The computer will NOT accept:

```
10 INPUT A INPUT B INPUT C
```

or

```
10 INPUT A, INPUT B, INPUT C
```

The colon can also be used to create open program lines for the sake of readability as already pointed out in the paragraph on clarity and readability.

To show it again:

```
10 INPUT A
20 :
30 B = A + C
```

The computer ignores line 20 during execution of the program, but the program listing (print on paper) has been made more readable.

The semi colon (;) is used to join together printed statements and strings, often in combination with the INPUT statment and the quotation marks:

```
10 INPUT "DATE OF BIRTH"; DA
```

Note the use of the quotation marks! When you RUN this program line, the computer will display DATE OF BIRTH? and a blinking cursor at the rght of the question mark. When you type in a date, it will be stored in the variable DA, thanks to the semi-colon. The semi-colon is mandatory in this case!

As a separator with the PRINT statement, it works just as the comma, with the major difference, however, that where the comma leaves considerable space between two results, the semi-colon narrows this to just a few spaces. Try this:

```
10 INPUT "DAY OF BIRTH";DA
20 INPUT "MONTH OF BIRTH"; MO
30 INPUT "YEAR OF BIRTH";YE
40 PRINT DA;MO;YE
```

The semi colon gives a very useful formatting effect. Try this:

```
10 A = B + 1
20 B = A
30 PRINT A;
40 GOTO 10
```

This is the same little program as at the beginnng of Chapter 1, but with a subtle difference. RUN it and see what happens. You will be surprised to discover what the addition of the semi-colon and colon in line 30 will do!

The QUOTATION MARKS (" ") are very useful tools when you want to display words, names, messages, etc. on your screen (and later on paper via a printer). They will also help you when you want to store a combination of LETTERS, such as names, into a variable (in this case a string variable which we will come to later).

You have already tried out some applications. See the programs in the paragraph on variables. Now if you want to display a word on the screen, you type:

```
10 PRINT "THIS IS AN EXAMPLE"
```

The program that converts feet into units of the metric system shows extensive use of the colon, the semi-colon and the quotation marks. Above all, note that when any character is placed between quotation-marks, it loses its operating function! When you RUN:

```
10 PRINT "THIS IS AN EXAMPLE: A + B = C"
```

the computer will display on the screen:

```
THIS IS AN EXAMPLE: A + B = C
```

Finally, when you want to store letter combinations (which we call STRINGS) you have to put them between quotation marks again, and place them into variable names which have the dollar sign ($) added to them.

For example, if you want to store names of cities it can become something like this:

```
10 A$ = "AMSTERDAM"
20 B$ = "BERLIN"
30 C$ = "COLOGNE"
40 D$ = "DOVER"
50 PRINTA$,B$,C$,D$
```


REM STATEMENT

As already pointed out, the REM statement stands for REMark. Its sole purpose is to comment on the programs you are about to write. Its function as a command is nil, but within the framework of a structured program it is very important, because it provides the necessary documentation.

It is said that a REM statement can be placed anywhere in a program. That is not entirely true. For example, the following program line won't work:

```
10 A = 2: REM THIS LINE WON'T WORK: B = 4 * A: PRINT B
```

RUN this line, and the machine replies with READY only. Because it does not go beyond A = 2. Whenever the computer encounters a REM statement, it will ignore what comes after it. The following will work:

```
10 A = 2: REM THIS WILL WORK
20 B = 4 * A: PRINT B
```

EXERCISES

Okay, you are now ready to write your first program, armed with the knowledge you have just acquired. I will close this chapter with a working example, and at the end you will be presented with a small assignment to work out.

```
10 REM CONVERSION OF POUNDS INTO ANY
20 REM UNIT OF IMPERIAL WEIGHTS
30 INPUT "ENTER NUMBER OF POUNDS";LB
40 :
50 SE = LB / 14:REM CALCULATION STONES
60 OU = LB * 16:REM CALCULATION OUNCES
70 GR = OU * 16:REM CALCULATION GRAINS
80 :
90 PRINT       :REM PRINT BLANK LINE
100 PRINT "IN STONES: ";SE
110 PRINT "IN OUNCES: ";OU
120 PRINT "IN GRAINS: ";GR
130 END
```

ASSIGNMENT

When you know the following:

1 lb = 453.6 grams

then write a program that converts stones, pounds, ounces and grains into grams. Refer to the example above. For a possible solution, see the Appendix.

CHAPTER 3

Flowcharting helps!

This chapter introduces you to the very helpful programming aid of flowcharting. At the same time we continue our discussion on some BASIC programming tools. You will be introduced to the IF-THEN statements, because they are most easily explained with the help of flowcharts. Closely connected with the IF-THEN combination are the GOTO and GOSUB commands which therefore will receive extensive treatment as well.

THE CONCEPT OF FLOWCHARTING

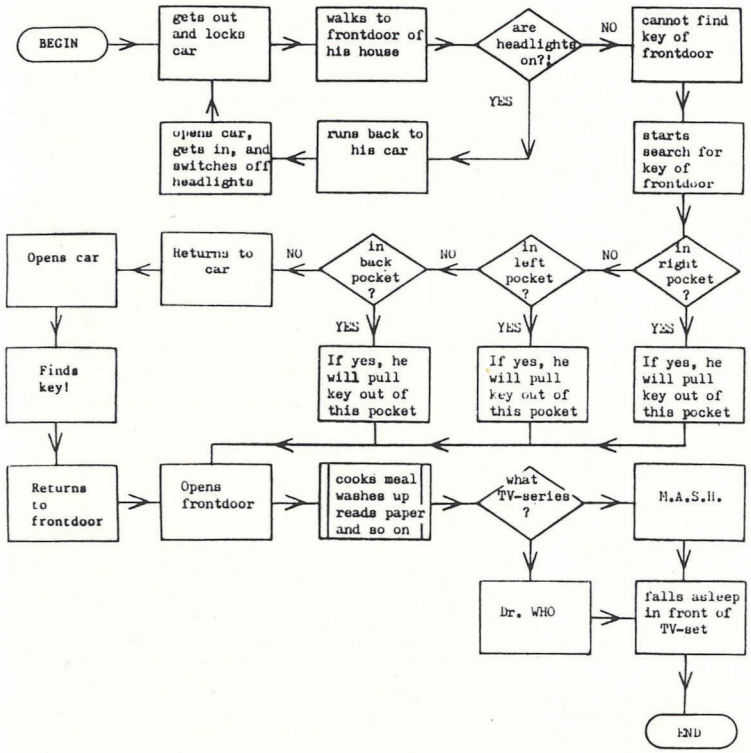
Have you ever realized that your entire life is in actual fact a sequence of decisions and actions evoked by these decisions? Most of us are not conscious of it, but from the very moment we awake in the morning, get up, have breakfast, go to school or work, and so on, we follow a more or less fixed pattern. We work off a program, so to speak.

I will now use such a pattern of everyday actions to explain to you the concept of flowcharting. It is about an hour in the life of Mr. Jones, first described in words only, and after that set out in a flowchart, thus depicting in graphic detail the entire sequence of events, decisions and actions.

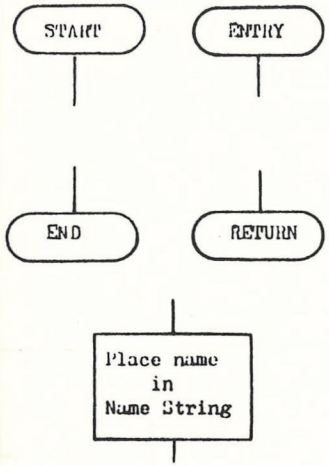
Mr. Jones is a happy-go-lucky bachelor, just coming home. He stops his car, gets out and locks the car door. But halfway to his front door he suddenly wonders, 'Are my headlights still on?' He turns round, and indeed the headlights are still switched on; he goes back, grumbling, opens the car and switches off the lights. He locks the car again and walks back to the front door of his home. But there he is confronted with a new problem, 'Where is the key?'.

It must have dropped off the key-ring. He can't remember whether he found it or not; he tries out all three of his pockets. But then he remembers that the house-key once dropped off the key-ring while he was driving. That could have happened now also. He goes back to the car, opens it, and indeed, finds the house-key on the floor of the car.

Relieved he locks up the car again, returns to the front door of his home and enters without any problem. Once inside he decides on a simple pizza, gets it from the freezer and bakes it in the oven. After finishing his meal and washing up (sometimes he likes to be tidy), he decides to watch a TV program. What will it be? M.A.S.H. or Dr. Who? He finds M.A.S.H. a better choice this time, but falls asleep in front of the TV-set.



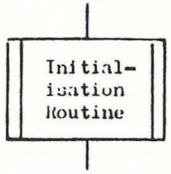
FLOWCHART SYMBOLS EXPLAINED



In the flowchart depicting 'an hour in the life of Mr. Jones', three symbols are used.

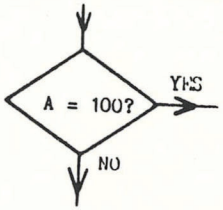
This symbol, called 'terminal', is always placed at the beginning of a flowchart and at the end. When it is used at the beginning, you can write in it 'begin', or 'start', and when it is at the beginning of a subroutine (refer to chapter 1 for subroutines) you can write in it 'entry'. At the end of a program you can write in it 'end' or 'stop' (however, 'stop' is not recommended — see about the command STOP in a later chapter). When you use this symbol at the end of a subroutine, you write in it 'return'.

A rectangle stands for an action, or as we say it in computer jargon, a 'process'. A statement like 'he gets out and locks the car' (see flowchart of Mr. Jones) is indeed nothing else but a description of an action. The purpose of a flowchart now becomes clear: you depict a process within such a flowchart as a rectangle, and later on this description must be translated into the language the computer understands.



A rectangle with double lines on the sides is a slight variation of the above. It indicates a 'predefined process' which simply means that what is written within this symbol indicates an entire program on its own, i.e. a subroutine. Whenever you encounter such a symbol in a flowchart, it means that a separate flowchart depicting the process described may have been drawn up elsewhere.

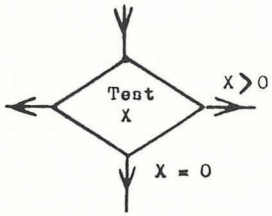
Note again the flowchart of Mr. Jones. There is one such symbol in it and it says 'cooks meal, washes up, reads the newspaper, and so on'. An entire set of actions is in one block, therefore requiring a separate flowchart, or in programming practice, a subroutine. In a following chapter you will see how you will use this symbol to your advantage.



This diamond shaped symbol is very important, because it refers to the true power of computers, that is the ability to make DECISIONS. Whenever you encounter this symbol in a flowchart, you are faced with a choice. It means that you, the programmer, must program in such a way that the computer will

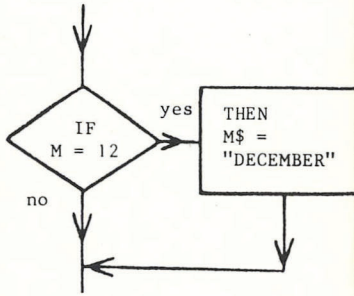
make a decision. Depending on the choice, the program flow will go in either direction, indicated by 'YES' or 'NO'. You will see this symbol most often in parts of a flowchart where the program is required to TEST A CONDITION.

Refer again to the flowchart of Mr. Jones. The first decision he comes across is the answer to the question whether the headlights of his car are still switched on, yes or no. IF so, the condition to be tested 'are the headlights still on?', has been met. The answer is YES and we say then that the CONDITION IS TRUE.



In that case, the program flow returns to the beginning of the program. Or, as we say it, the program flow BRANCHES OUT. (Which is why this symbol is used in situations which we call 'CONDITIONAL BRANCHING'). Okay, let us return to Mr. Jones.

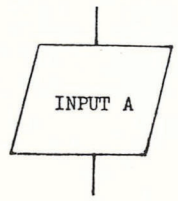
The condition is found to be true, so he returns to his car, and switches off the headlights. When he walks back to the front door he could ask himself again, 'Are the headlights still on?'. Then the answer will be NO. The condition now tested proves to be untrue, or as we prefer to say, it proves to be 'false'. The consequence is that the program flow does not branch out, but follows the direction it is supposed to go.

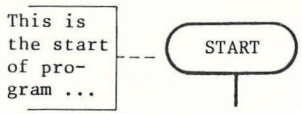


Now you must realize that I used the word 'IF', in capital letters. I did this because it is in program situations like these — where conditions are tested — that the IF-THEN combination proves to be most useful. More about that later.

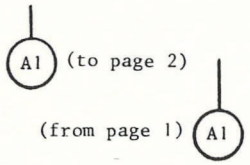
A FEW MORE FLOWCHART SYMBOLS

The parallelogram symbolizes Input or Output. Hence it can be used to indicate where in the program you will use some form of input, for example by implementing the command INPUT. It is also used to indicate some form of output, i.e. on the video-screen or on paper, via a printer. As this symbol stands for many functions within the range Input-Output, it is used to indicate what kind of Input or Output it is going to be.





This is the so called 'annotation' symbol. Often there is not enough room within the boxes to write the description of the process you have in mind. In that case you can use the annotation symbol to add extra comments.



A small circle is a 'connector'. Sometimes flowcharts can become rather complicated (always try to avoid that!). One way to make a flowchart not resembling a spiderweb is the use of connectors instead of long flow-lines. Then there is one connector at the sending end and another at the receiving end. Both connectors must have the same code placed in them to indicate precisely where the flow of the program has to go.

There are many more flowchart symbols available, but with the ones shown here you can do most of the work.

Naturally, flowcharts can be drawn up by hand on a piece of scratch-paper. But obviously this will soon degenerate into a mess. I therefore strongly suggest you purchase a flowchart-template. Probably your computer dealer sells them, and if not, you can go to a supplier of computer stationery or a supplier of artists materials.

FLOWCHARTING PROS & CONS

As with anything, there are always two sides to a coin. This applies to flowcharting — it has clear advantages, but also quite valid disadvantages.

Let me emphasize first that the importance of flowcharting depends strongly on the computer language it is used in conjuction with. In contrast to BASIC, many other languages are NOT interactive. In the latter case elaborate flowcharts are considered a MUST before the programmer even dares to touch the computer. Languages which are not interactive do not leave much room for extensive testing. The programmer 'plays computer' with the flowchart instead — to test a possible program flow, before he writes the code in the syntax of the computer language.

With BASIC, on the other hand, you can test continuously what you have written. Therefore many

BASIC programmers do not feel the need for flowcharts. To a large extent they are right, to a minor extent not. Flowcharting cannot be dispensed with totally, because then we can't profit from its distinct advantages:

(a) the prime advantage of a flowchart is that it makes highly visual what you plan to write;

(b) flowcharting can be very helpful in the solving of difficult problems. For example, in the analysis of intricate parts of a program, specifically those where a lot of decisions must be taken (this will become clear when we discuss the IF-THEN combination);

(c) an overall 'rough' flowchart will provide an excellent guide in the setting up of the mainline and the consequent subroutines (see next chapter).

It is strongly recommended you work on the basis of such an 'overall' flowchart when you are in the initial stages of a program design. This fits in very well with the rules of structured programming.

What about very detailed flowcharts? It is here this book deviates from general opinion because this is where the disadvantages of flowcharting become clear.

Flowcharts are very useful in the design of small programs, small subroutines, and testing of intricate program flow patterns. A small flowchart fitting on one A-4 piece of paper can indeed provide a quick insight into a programming problem — on the condition, however, that it has been drawn up nicely.

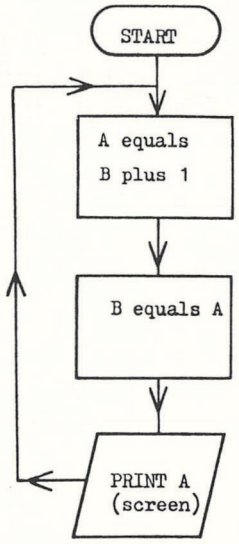
It must be asserted here that many programmers are not particularly talented when it comes to the drawing up of open, easy-to-follow flowcharts. It becomes much worse when a large program has to be flowchart documented over many pages. There is not much pleasure in trying to read a program flowchart extended over twenty or so pages. In short, large flowcharts tend to become a bunch of spaghetti, and so programmers prefer to ignore them, even in the case where existing programs must be updated. So a well written, structured program, which includes clear comment lines, is easier to follow than a large flowchart.

Obviously the drawing up of large flowcharts is a very tedious and time consuming process, particularly when one wants to do it as well as possible. Then, when a program is updated, the flowchart should be updated as well. That implies redrawing, and it is quite natural that one tends to forget this. Hence after some time the flowchart does not entirely resemble the original program.

On the other hand, when a programmer updates a program it is easy to insert appropriate comment lines. Good program design, that is structured programming, following the top-down approach, can be done without extensive flowcharting. Therefore I recommend flowcharting only in a limited form, that is essentially as a programming aid. In its limited form, as explained some paragraphs ago, it can be very helpful; in its extended and detailed form it can become an unnecessary burden. In general I would say, consider flowcharting as optional.

IF-THEN

Remember our discussion of the endless loop (chapter 1). Here again is that little program that does the simple thing of counting.



You see clearly the idea of a loop in this program depicted in the flowchart, while that is less obvious in the four program lines. As a matter of fact, it is an endless loop. It goes on for ever and ever when it is started. It can only be interrupted by either switching off the computer (and then losing the program), or by pressing the RUN/STOP key. But that is not an elegant solution.

It is much better to build an interrupt into the program itself. We do that with the previously mentioned technique of conditional branching. In the program a condition is tested and when the condition is 'true', then the program will branch out and come to a stop. Now you will see the use of the IF-THEN combination:

```
10 A = B + 1
20 B = A
30 PRINT A
40 GOTO 10
```


What happens in the second example is that with every loop, the number stored in A is tested in line 40. The test is simply "does the number stored in A equal 100". IF the condition is found to be true (the answer is yes), THEN the computer directs the program flow to line 60. Or as the line says: THEN 60. The program branches out at this point. But, as long as the condition is NOT met — the condition is found to be false — the program will jump to the next line (50). To get a full understanding of what happens here, I suggest you type in the program and RUN it. Perhaps you could also alter line 40 a few times, for example by IF A = 25 THEN 60, or IF A = 1000 THEN 60, and so on.

You must realize that the combination IF-THEN is actually read as: IF (now follows a conditional statement) THEN (now follows a command that directs the program flow elsewhere when the condition is found to be true).

Between IF and THEN you can place any statement which includes a 'relational operator' (you know them: = , > , < , = > and = <), while after THEN may come almost any other statement or command valid within the rules of the BASIC language. A few examples later on will explain this.

By the way, the line

```
40 IF A = 100 THEN 60
```

can be changed into

```
40 IF A = 100 THEN GOTO 60
```

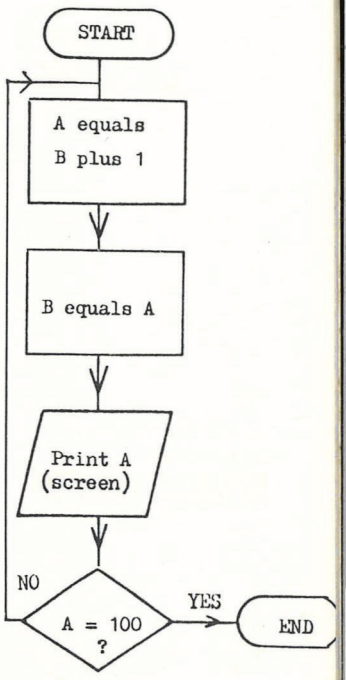
Your computer understands both.

As for the effect of the total program, you will obtain the same result when you change line 40 this way:

```
40 IF A = 100 THEN END
```

When the condition A = 100? is found to be true the program will end in line 40 instead of 60 in the other cases.

However, easy as it looks, this way of ending a program is not recommended, because it does not fit into the framework of structured programming. The END statement, just in the middle of nowhere so to speak, may do the job of stopping a program, but makes the program itself unclear.




```
10 A = B + 1
20 B = A
30 PRINT A
40 IF A = 100 THEN 60
50 GOTO 10
60 END
```

Here are a few extra lines, showing valid uses of the IF-THEN combination:

```
100 REM ESTABLISH MONTH
110 IF A = 10 THEN MO$ = "OCTOBER"
200 REM CIRCLE FUNCTION
210 IF B = > 360 THEN B = B - 360
300 IF C=10 AND D=20 THEN GOSUB 10
```

NOTE: the last line employs the 'AND' operator and the GOSUB command. More about the AND operator in a later chapter.

THE GOTO COMMAND

You have come across the GOTO command several times now, so you won't have many problems with this one anymore. Its function is quite obvious. The GOTO command directs the program flow to another part of the program indicated by the line number placed after it. When GOTO is used in conjunction with the IF-THEN combination, then we speak of conditional branching. When the GOTO command is used as a stand alone command, we speak of UNconditional branching. Just have a look at this program:

```
10 INPUT A
20 INPUT B
30 C = A + B
40 GOTO 100
50 D = A - B
```

In line 40 the program encounters the GOTO 100 command and therefore jumps to line 100 without any testing. In a case like this all the program lines between 40 and 100 are of course rendered useless, unless somewhere after line 10 another GOTO command directs the program back to line 50.

By the way, it is a general consensus in the world of programming that excessive use of the GOTO command is often a sign of poor programming techniques. There are more elegant techniques of branching available, as you will learn in due course.

By the way GO TO (thus a space between GO and TO) is also allowed.

GOSUB RETURN

When you know the ins and outs of GOTO, the next subject to study is the combination GOSUB RETURN. GOSUB stands for GO to SUBroutine and must be followed by a line number. The function of RETURN will become clear soon.

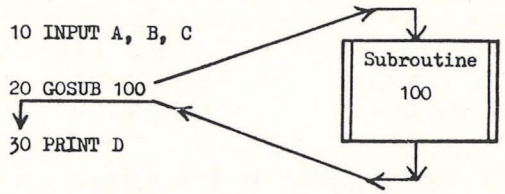
The meaning of GOSUB can be best explained by an analogy with everyday life. Whereas GOTO means 'go to a designated spot and remain there unless directed otherwise', GOSUB means 'go to a designated spot, complete a specified task, and then RETURN immediately, and wait for new orders'.

It is something like doing a carpentry job, and discovering that you have run out of nails. You start out with 100 nails and discover after hammering in all of them, that you have not got enough. Your program directs you now to the hardware shop, where you buy another 100 nails, and then you RETURN immediately to proceed with the job. Again after some time you run out of nails and go back to the hardware shop, buy another 100 nails, and RETURN immediately home to proceed with the job.

Now regard that hammering job as the program, and the intermission to the hardware shop as the subroutine. Naturally this is an analogy which does not portray exactly the function of a subroutine. It shows a very inefficient carpenter, whereas the purpose of subroutines in a program is the ENHANCE EFFICIENCY! But the analogy gives you an idea of how it works: go to a designated place and return immediately afterwards. In a program it would work like this:

```
10 INPUT A,B,C:REM INPUT HRS MNS AND SECS
20 GOSUB 100 :REM CONVERT TO DECIMAL HOURS
30 PRINT D   :REM PRINT DECIMAL HOURS
```

What happens in this program is that the program is directed in line 20 to a subroutine where apparently the variables A, B and C containing hours, minutes and seconds, are combined and converted to decimal hours, stored in variable D. When this SUB-task has been completed, the program returns to line 20 but then proceeds immediately to line 30.The diagram will clarify this.



But now remember this: the GOSUB command only works correctly when the subroutine itself ENDS WITH A RETURN STATEMENT!

In our example subroutine 100 could look like this:

```
100 REM THIS SUBROUTINE CONVERTS . . . etc
    o
    o   (here processing takes place)
    o
150 RETURN
```

NOTE: a subroutine MUST be closed by a RETURN statement. If not, the computer will react with a GOSUB WITHOUT RETURN ERROR!

As stated, subroutines exist to enhance efficiency. To show you what I mean study the following.

TIME PROBLEM

If you have ever done arithmetic with hours, minutes and seconds, then you know that it is a confusing task, because you do not work with decimals but with 'sexagesimals', that is the division of a number into sixty units instead of ten. The problems imposed by this can be overcome by converting hours, minutes and seconds to decimal hours first before you do any arithmetic. Obviously this is something a computer program can do for you.

The way to convert a given set of hours, minutes and seconds to decimal hours is as follows. First divide the seconds by 60, add this result to the minutes, divide the now decimal minutes again by 60, and add the remainder to the hours. You then have decimal hours. A program-line which solves this problem could look something like this:

```
5 SE=SE/60:MN=MN+SE:MN=MN/60
:HR=HR+MN
```

Now let us write a complete program that adds three sets of hours, minutes and seconds (for example: 5h17m33s, 21h53m31s and 10h41m22s):


```
10 INPUT "  HOURS";A
20 INPUT "MINUTES";B
30 INPUT "SECONDS";C
40 C = C / 60
50 B = B + C
60 B = B / 60
70 A = A + B
80 INPUT "  HOURS";D
90 INPUT "MINUTES";E
100 INPUT"SECONDS";F
110 F = F / 60
120 E = E + F
130 E = E / 60
140 D = D + E
150 INPUT"  HOURS";G
160 INPUT"MINUTES";H
170 INPUT"SECONDS";I
180 I = I / 60
190 H = H + I
200 H = H / 60
210 G = G + H
220 K = A + D + G
230 PRINT K
240 REM OUTPUT IN DECIMAL HOURS
```

Now note this fact. This program has three complete sets of input, respectively lines 10, 20, 30, lines 80, 90 100, lines 150, 160 and 170. All of them do essentially the same work. And also consider lines 40, to 70, lines 110 to 140, and lines 180 to 220. Again, these three sets of routines do essentially the same thing. The program will work out fine BUT IT IS HIGHLY INEFFICIENT!

```
10 GOSUB 100
20 A = X
30 GOSUB 100
40 B = X
50 GOSUB 100
60 C = X
70 D = A+B+C
80 PRINT D
90 END
```

```
95 REM SUBROUTINE
100 INPUT"HOURS  ";X
110 INPUT"MINUTES";Y
120 INPUT"SECONDS";Z
130 Z = Z / 60
140 Y = Y + Z
150 Y = Y / 60
160 X = X + Y
170 RETURN
```

HOW IT WORKS

On the left is the 'mainline' of the program (lines 10 to 90), and at the right you see the subroutine (lines 100 to 140). It is that one subroutine which does the work. Just follow the program lines. At 10 there is the command GOSUB 100. Hence the program pointer jumps to line 100 — there it enters the subroutine. In this subroutine hours, minutes and seconds are converted to decimal hours, and at line 170 the program encounters the command RETURN.

The program returns to the mainline and re-enters this mainline at line 20. There it stores the result obtained in the subroutine as variable A. In line 30 the program jumps again to the subroutine, comes back at 40, and stores the result in variable B, and so on. This program is a lot more efficient than the previous one which does the same job. This is why the concept of subroutines is so important in computer programming. No large program will be without them.

PROGRAM FLOW IN WORDS

Elsewhere in this chapter we discussed the advantages and disadvantages of flowcharting, and found it to be optional at best. There is an alternative, however, which is becoming increasingly popular in the computer-industry. Some experts call it 'pseudo-code', but here we prefer the term 'program flow in words'. Such a program flow in words can also clearly outline the proposed program, which is why I conclude this chapter with a program flow in words relating to the program of the previous paragraph. Study it carefully and then decide for your preference — flowcharting or program flows in words.

Mainline

START Mainline
  Go to subr hr-mn-sc/decim. hrs
  Return to mainline
Store result of subr. in A
  Go to subr hr-mn-sc/decim. hrs
  Return to mainline
Store result of subr. in B
  Go to subr hr-mn-sc/decim. hrs
  Return to mainline
Store result of subr. in C
Add A to B to C
Store result in D
Print contents of C
End of program

Subroutine conversion hrs, mn, sec to decimal hours

Input number of hours, store in variable X
Input number of minutes, store in variable Y
Input number of seconds, store in variable Z
Divide seconds by 60
Add decimal decimal seconds to minutes
Divide minutes by 60
Add minutes to Hours
Return to mainline.

CHAPTER 4
Planning your program

APPLYING YOUR KNOWLEDGE

When you have thoroughly studied the previous chapters you will be able to design and write a rather simple, but good, working program. In this chapter you will apply the knowledge you have acquired so far (where the basic programming tools are concerned). We will discuss in detail the top-down approach as incorporated in the rules of structured programming.

It all boils down to the maxim, THINK BEFORE YOU PROCEED! Because the sooner you begin writing a program without planning it thoroughly beforehand, the longer it takes to complete a good job. A well designed and written program is the result of some 70% thinking and 30% writing (or CODING as programmers prefer to call it).

THE FIRST JOB

This chapter and the following one will be devoted to designing a program that will convert weights and measures of the imperial system into the metric system and vice versa. I have chosen this as a subject of computerized problem solving, not only because of its general appeal but also because the calculation procedures are very simple, and hence the programming procedures won't be difficult either. Moreover, a well functioning program can be written with the knowledge acquired in the previous chapters.

SUMMARY OF PROCEDURES TO FOLLOW

Let me summarize the procedures you will follow from now on (details will be explained later):

(1) Define and specify the job thoroughly — this includes research.

(2) Design the format (i.e. layout) of what you think is going to be the output of the program.

(3) Design the format of what you think will be the necessary input.

(4) Work out the number of procedure blocks (or 'routines' or 'modules' — we will come to that later) that you think will be necessary to complete the job.

(5) Work out the mainline of the program — preferably in flowchart form. Work out the following lower level procedures — also in rough flowcharts. NOTE: as explained before, flowcharting is optional. If you decide against flowcharting then WRITE down the sequence of procedures you think will be necessary.

You will find examples of written program flow concepts at the end of this chapter.

NOW THE CODING

So much for the design, now to the coding of the program:

(6) It is essential, within the framework of top-down programming, to start with the highest level routines, as outlined in your rough general flowchart, and then work on the next levels.

(7) The mainline is truly a mainline — it may not contain detailed programming routines. Detailed program routines are to be found in lower level modules.

(8) You only proceed to a next level when the present routine has been completed and verified — that is, you make sure that it works.

(9) Postpone refinements until the final stages of programming. The first priority is to get every level working properly, and consequently the entire program.

(10) When the computer allows it (i.e. it has enough memory available), do not hesitate to insert sensible comments (REMarks).

(11) Though it is not a truly essential part of the program, it is always handy to initialise a program with an 'identification section'. For the sake of clear documentation it is also important to include a section which describes the variables used in the program.

(12) This one refers particularly to the owners of printers — write your program in a nice format. When you print the program listing, it will make quite a difference!

We will now discuss every item in detail. (This chapter contains elaborate treatments of items 1 to 5, the next chapter contains items 6 to 12).

A SPECIFIC TASK

STEP 1: DEFINE THE JOB THOROUGHLY

Consider yourself as your own client. You are the programmer and you commission yourself a specific task which is to write a program that converts measures and weights of the imperial system into those of the metric system, and vice versa. It is your task to write a program that is not only understood by you, but also by your client! Now don't panic! Do not think that you are unable to do that — you will be able to, because it is a lot simpler than you may think.

On the other hand, if you are the optimistic type who says, 'Let's get on with the job at once, and I'll finish that within a few hours', then you are definitely wrong. Any programming job always takes longer than you think, particularly when you do not spend enough time figuring out what the job is all about.

IN MINUTE DETAIL

So before you touch the computer, define the job thoroughly, i.e. in minute detail. It is not enough to say that the program must convert weights and measures of the imperial system into those of the metric system. This is so vague that it can hardly be called a job definition. It will have disastrous consequences when you do not fully understand what is required by the client (even if the client is you, as is the case here). You may be well into the program and

then discover that you have forgotten..., or that you can better leave out..., and so on.

So let us now define precisely the present problem. When the requirements are 'converting weights and measures from imperial to metric and vice versa', then the first question is 'which weights and which measures?' So you have to specify clearly which weights and which measures are to be converted.

The next question may be, 'how do you want to have it converted?' That is do you want the option of e.g. typing in values for EVERY standard of measure of weight and then receiving in the output all conversions? Or do you want to type in one specific standard weight or measure?

TO CLARIFY

In the first option you can ask the program to solve the problem 'how many centimetres in 15 inches?' or 'how many yard in 100 metres?' or 'how many imperial tons are 10000 kilograms?' and so on. Of course, a program that delivers such output is quite feasible, but you must realize that it will be a more complicated program than the one which gives the other option — input of a STANDARD measurement or weight which then produces all conversions, both in metric and imperial. Let us opt for the last option. Hence the specifications are:

Re lengths, metres or feet as input, and as output all practical conversions in both systems.

Re weights, kilograms or pounds as input, and as output all practical conversions in both systems.

Even that specification is not enough. You have to define which standards of measurements and weights will be involved.

YOU NEED TO KNOW

Here another important part of the preparations must be memorized — RESEARCH! A problem must not only be properly defined, it must also be researched! A case like this illustrates this point clearly. If you want to have all these conversions done by the computer program, you need to know the exact formulae of the conversions you want to include. You have to go back to the books, and find out how many centimetres equal one inch, or how many pounds equal one kilogram, and so on. For that matter, I have done that for you already. Here are the next specifications:

TABLE OF WEIGHTS AND MEASURES (imperial to metric)

(weight)				
1 imperial Ton (T)	=	20 cwt (hundredweight)	=	1016 kg
1 hundredweight (cwt)	=	4 quarters)	=	50.8 kg
1 quarter (qr)	=	2 stones	=	12.7 kg
1 stone (st)	=	14 pounds (lb)	=	6.35 kg
1 pound (lb)	=	16 ounces (oz)	=	453.6 grammes
1 ounce (oz)	=			28.35 grammes
1 kilogramme	=	1000 grammes		
1 hectogramme	=	100 grammes		
1 decagramme	=	10 grammes		
(length)				
1 nautical mile (knot)	=	2025 yards	=	1852 metres
1 statue mile	=	1760 yards	=	1609.315 metres
1 yard	=	3 feet	=	0.9144 metre
1 foot	=	12 inches	=	0.3048 metre
1 inch	=	1 inch	=	0.0254 metre
1 kilometre	=	1000 metres		
1 hectometre	=	100 metres		
1 decametre	=	10 metres		
1 metre	=	100 centimetres	=	1000 millimetres
1 decimetre	=	10 centrimetres	=	100 millimetres
1 centimetre	=	10 millimitres		

With these tables you can work out precisely the conversions you have in mind. But moreover, you will decide now, which standards you will use in the program. I have already decided for you. All standards as set out in the table above are included in the program.

A SUMMARY

So let us summarize the specifications of the job:

(a) Convert measurements (length) and weights from the imperial system to the metric system and vice versa;

(b) Input standards: feet and metre for length; kilogram and pound for weight;

(c) Output both in metric and imperial, for either section (measurement or weight).

A last remark before closing this subject. Of course you will write programs mainly for yourself. But even then, avoid making programs which are only understood by yourself. A good program is one which, amongst other things, can be easily understood by others when it is RUN. Never assume that others will understand. Never think 'this is obvious'. What is obvious to you is quite often totally unclear to others. It is because of this that many programs, which are essentially good, are a commercial disaster because they are not 'user-friendly'. That means, a program must be friendly, i.e. easily accesible, to any user, and not only to its author.

STEP 2: DESIGN YOUR OUTPUT FORMAT

The reason why I put the planning of your output next in line to the job definition, is because it is part of the job definition. The main objective of every program is what it must deliver, a result, i.e. as you know output can be presented in two ways, on paper or on screen. As for the first we often speak of 'hard copy'. Whatever the case, your output must be understood. It must clearly state what a certain figure, printed on screen or on paper, actually represents. If, for example, you write a program that looks like this:

```
10 INPUT A
20 G = A * 453.6
30 PRINT G
```

and consequently RUN it, then it will be totally unclear to anyone else what is happening here. You put in the digit 1, for example, and the result, as presented on the screen is 453.6. The question now is 'what was the input', and as for the output '453.6 . . . what is it?'

It will be a distinctly different matter when I rewrite this program this way:

```
10 INPUT "POUND ";A
20 G = A * 453.6
30 PRINT"EQUAL ";G;" GRAMMES"
```

The other reason that you must plan your output carefully before you start coding the program is that an output format tells you exactly what goal you have

to work towards, when you design and code your program. But there are two ways to go about an output format. The first one is the no nonsense approach, just print the results with all necessary extra information on the screen, and begin at the far left side. For the sake of clarity print some blank lines now and then. This is alright when:

(a) Not too many results are to be printed, and

(b) When you don't care about a 'pretty presentation'.

However, realize that a pretty presentation, if not overdone, helps to get the message across, whereas the no nonsense approach may fail on that point. Therefore a pretty presentation is highly preferable. The difference between the two is easily demonstrated by the two examples below:

```
 100000 METRES EQUAL:

MILLIMETRES   100000000
CENTIMETRES    10000000
 DECIMETRES     1000000
     METRES      100000
 DECAMETRES       10000
HECTOMETRES        1000
 KILOMETRES         100

      INCHES   3937007.87
        FEET    328083.99
       YARDS    109361.33
       MILES   62.1371192
       KNOTS    54.005595
```

A neat output format

```
 100000 METRES EQUAL:

MILLIMETRES 100000000
CENTIMETRES 10000000
DECIMETRES 1000000
METRES 100000
DECAMETRES 10000
HECTOMETRES 1000
KILOMETRES 100
INCHES 3937007.87
FEET 328083.99
YARDS 109361.33
MILES 62.1371192
KNOTS 54.005595
```

An ugly output format

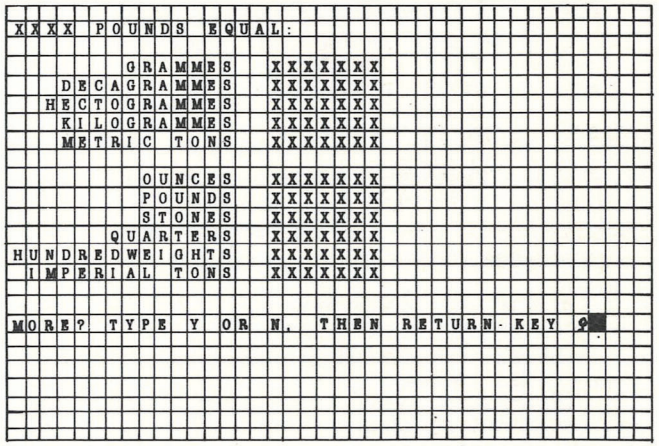
SCREEN FORMAT

How do you set up a nice output design? On paper, of course, before you even touch the computer. Use graph paper with, for example, 5 mm squares. Then outline a rectangle which simulates your screen. You must take into account the screen format of your computer. Your Commodore 64 gives a screen of 40 squares wide by 25 squares high.

It may be very useful to go to a copying service and have quite a few of those screen layout worksheets copied. You will need them!

First write on a piece of scratch paper a rough layout of what you have in mind. Then count the letters of each word, and when it is a sentence, do not forget to include the spaces between the words. Realize that words and sentences have to fit into the width of 40 squares, and when you have a large block of text, do not let it exceed a height of 25 lines. Fiddle around with the words and sentences on your graph paper till you are satisfied with the output format. Below you will see what the output format of our conversion program might look like:

OUTPUT FORMAT



This picture clearly shows how you can vertically line up words of unequal length; you count the length of the longest word, here 'hundredweights', which has fourteen letters. Then you make sure that all other words get extra spaces added to them, so that the words plus spaces make fourteen in total.

```
PRINT "        GRAMMES"
```


STEP 3: DESIGN YOUR INPUT FORMAT

What I just said about output design also applies, to some extent, to the design of the input format. (To some, that is a lesser extent, because input is the starting point of the program. It is not the goal of the program). You are not doing anything wrong when you work out the input format before you start the overall design of the program.

After all, the input format MUST TELL the user EXACTLY what is required as input. There is a proverb in the world of computer programmers which is called 'GIGO'. It stands for 'Garbage In, Gargabe Out'. This simply means that when you put in wrong data, the program will deliver a wrong output. Which is why a clear, i.e. correctly designed, input format is crucial to the proper functioning of any program.

USER FRIENDLY

I have already hinted at the 'user friendliness' of a program. It not only refers to program output but perhaps more so to input. When any person, other than the programmer, finds it difficult to figure out what is required as input, then the program is not user friendly. Of course, programs of a specialist nature do not have to be understood by everyone — but if it is not understood by the specialist for whom it has been written, then something is definitely wrong!

The following two examples will clarify this.

(Example 1)

```
10 INPUT DA
20 INPUT MO
30 INPUT YE    etc.
```

When this input section of a program is RUN no one, except the programmer, will understand what input is required, because the computer displays question marks only. Obviously this is poor input design. But it will be a different matter in example 2.

(Example 2)

```
10 PRINT "INPUT BIRTH-DATE"
20 PRINT                    :REM PRINT BLANK LINE
30 PRINT " ENTER DAY NUMBER, THEN MONTH
   NUMBER, THEN YEAR"
40 PRINT                    :REM PRINT BLANK LINE
50 INPUT "DAY "; DA
60 INPUT "MONTH"; MO
70 INPUT "YEAR"; YE         etc.
```

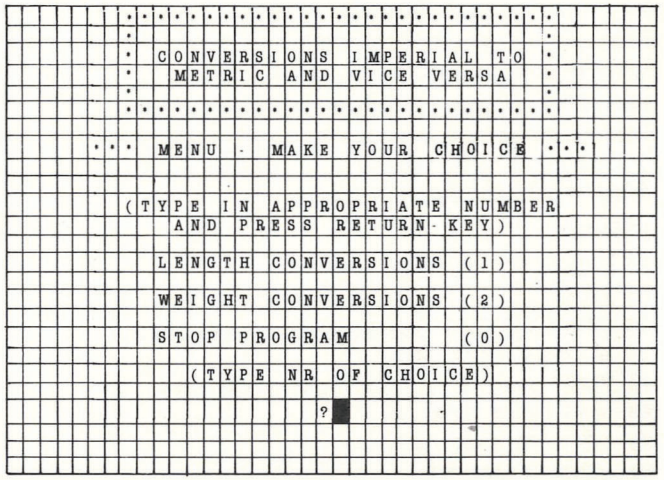
This input routine shows clear screen messages and requests (or 'PROMPTS' as we prefer to call them, because they prompt the user to input data).

THE JOB MENU

Programs which particularly offer a choice of tasks need to have a 'job menu' in their input format. This simply means that the input format clearly states the different options the program has to offer. In our example program 'conversions' there are two options. Hence there are two possibilities on the menu: (1) length conversions and (2) weight conversions.

As you will see later it is not at all difficult to write a routine which directs the program by 'prompts' as indicated in the job menu. A simple, and very common way, to initialise such an option is hitting a letter or digit key which refers to that option. In our sample program the option 'length conversions' can be referred to by the number 1 for example, and the option 'weight conversions' can be referred to by the number 2. Hence the 'prompt' for the first option may be something like 'TYPE 1 AND HIT RETURN FOR LENGTH CONVERSIONS'. And something similar but then using 2 as the prompt number for weight conversions. There are, of course, several variations possible. See below the input format which was designed for this program.

INPUT FORMAT



NOW NOTE A FEW THINGS

The top part of the screen is occupied by a box made up of asterisk symbols containing a message of what the program is all about. This screen message acts as an identification. It is never wrong to include an identification message in your input format.

There is a third option, 'STOP PROGRAM' referred to by the number 0 (zero). Although the easiest way to stop a program is pressing the RUN/STOP key or just switching off the computer (you then wipe it off completely, of course), it is not very elegant. A good program contains a 'termination routine', which is therefore included in the job menu.

STEP 4: WORK OUT THE POSSIBLE NUMBER OF PROGRAM MODULES

An essential part of the top-down approach is that you work out how many separate program procedures may be necessary to complete all the tasks the program is supposed to perform, before you ever start writing the program. We have some names for these program procedures: subroutines, subprograms, and modules.

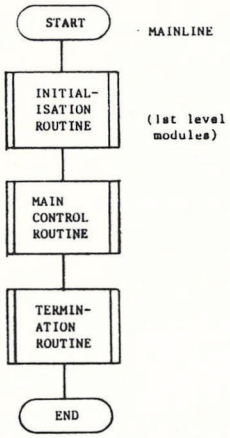
To avoid confusion with subroutines which have to be addressed by the GOSUB command, it is preferable to talk about 'routines' only, i.e. with names attached to them describing what task the routine has been assigned to. For example; 'weight conversion routine', 'length conversion routine', 'main control routine', and so on.

MODULES

In more general terms we can talk about 'program modules' or just 'modules'. Then we can say something like 'the next module to write is the Length Conversion Routine'. Ideally every module should be a stand alone program, in the sense that it can function separately and independently of other modules. In reality this is of course not entirely feasible, because the proper functioning of a module depends heavily on what it receives from other modules as input.

But that is not relevant as regards the principles of top-down programming; the idea is simply that a module can work despite other modules. This will be expanded in the next chapter. The idea of top-down programming is that stand alone modules are linked up in a specific order — from first (= the highest) level to lower levels (levels 2, 3, 4 and more if necessary). Here I refer to the last paragraphs of chapter 1 which gives a clear outline of the principles pertaining to this subject.

MAINLINE



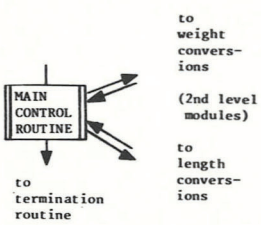
First of all, every structured program has a MAINLINE. This mainline can be a link up of just a few modules or many, all depending on the complexity of the program. This 'conversions' program is in essence very simple, hence the mainline can become simple. Every mainline has an initialisation module, and a termination module (see also the paragraph on the formatting of input). Between these two we can place a module which controls the actual program. Therefore we call it the 'main control routine'.

Let us concentrate on the general ideas pertaining to this mainline. The first module is the Initialisation Routine. This module can consist of two parts, an Identification section and a section which initialises all kinds of program functions and constant variables. That second section is often quite vital. Depending on the complexity of the entire program, this section will contain DIMension statements, and DEFine FuNction statements. We have not discussed these BASIC programming tools as yet — that will follow in a later chapter. Just keep it in mind. As for constant variables, that is quite simple.

AN EXAMPLE

If for whatever reason, you need the famous constant of PI in your program, you can define it in this initialisation routine like this, PI = 3.141592654. In our example program we do not need this initialisation of program functions and constant variables.

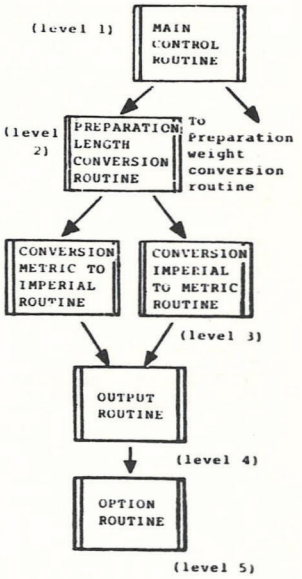
The last module of the mainline is, in this case, just a formality. It stops the program, it brings the computer to a standstill. However, in other programs the termination routine may play a vital part as it may contain the final output. It is the second module which is the truly important one of our example program, because it controls the entire program flow. It receives the input initialisation and depending on that information it will direct the program flow to one of the SECOND level modules.



When a job has been completed the program flow RETURNS to this Main Control Routine. Remember that one of the rules of top-down programming is that the program flow always returns to the mainline when a specific job has been completed.

Let us now follow the program flow when it branches out from the Main Control Routine.

You see here what happens when you have prompted the program flow to branch out to the next level of the program. The program flow enters one of the second level modules, here the Preparation Length Conversion Routine. Why preparation? Well, there are two ways to convert lengths:

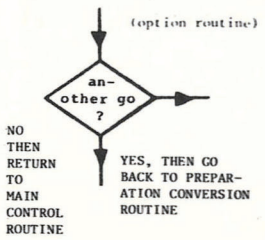


(1) from Metric to Imperial, and

(2) from Imperial to Metric.

This module leaves the choice to the user, just as in the Main Control Routine. When the choice is made, again by prompts, the program flow will go to the next, i.e. a third level module. The actual conversion takes place here. The user is prompted to enter a number, then the routine converts it into the appropriate standards of the other system, and when that is done, the program flow moves to the next module, which again is in a lower level, the output routine. This routine produces a nice output on the screen (at least, that is your intention).

Is that the end? Yes and no. Yes, because the program gives you what you wanted. However, for one program RUN only. That is why I now bring in 'no', because finishing a program in this crude way is not only inelegant, but also unsophisticated, because the program is rather ineffective.



Every time you want a conversion, you have to re-RUN the program, when you let it stop after the output. (Besides, it is against the rules of structured

programming. When the program flow has gone through lower level routines, it should return to the mainline eventually.) The program becomes much more effective when you add another module, which gives you an extra option. 'Do you want another go (i.e. do you want more conversions e.g. length or weight — it just depends which line you have chosen in the Main Control Routine) Yes or No?'

In the first case, a simple prompt can return the program flow to a Preparation Conversion Routine of the conversion stream you are in. In the second case the program flow goes back to the mainline, in fact to the Main Control Routine, where again you are faced with choices.

We have now discussed in rough detail the entire program flow of one RUN. And as you have seen this program flow has been portrayed in rough flowcharts consisting of modules. However, as I pointed out at the beginning of this chapter, you can also give a WRITTEN OUTLINE of a possible program flow.

This program flow in words can be built up just as easily in modules as we have done with flowchart symbols. What you need here is some verbal skills in the sense that you have to write down precisely what you want the program to do.

TO CONCLUDE

We now know how many modules for one particular topic of conversion are needed.

The mainline (level 1) contains three modules. The range of routines which work out the length conversions contain one level 2 module, two level 3, one level 4 and one level 5 module. As the conversion of weights requires basically the same sort of calculations it is obvious that the range of routines which work out weight conversions is essentially the same as the range of length conversion routines. This is why the module flowchart shows a remarkable symmetry.

A last remark. Ideally these preliminary designs of the program we are going to develop should also be the final designs. Programming, however, is just like a game of chess: only the very clever can see every move ahead. So there is a good chance that when you finally set out the coding of the program, it is not at all unusual that you will change your mind, and impose alterations.

It even happens that you may decide to start all over again. That is nothing to be ashamed of. These are the things which make programming such a creative thing to do.

It is for this reason that the program which comes from this initial design may quite deliberately differ at some levels from that initial design. But the overall idea has been maintained throughout.

PROGRAM FLOWS IN WORDS ONLY

Mainline
Start
Go to Initialisation Routine
Go to Main Control Routine
Go to Termination Routine
Stop

From Main Control to Output and beyond
If decision in Main Control for Weight Conversions, go to Preparation Weight Conversion Routine
If decision in Main Control for Length Conversions, go to Preparation Length Conversion Routine
(assume decision is for Length Conversions)

Preparation Length Conversion Routine
If decision in Preparation Length Conversion Routine for Metric to Imperial go to Input of Metric data
If decision in Preparation Length Conversion Routine for Imperial to Metric go to Input of Imperial data

(assume decision is Metric to Imperial)

Input Metric data
Type in Metric data (in Metres)
Calculate results in Metric standards of length
Calculate results in Imperial standards of length
Go to Output Routine

Output Routine
Print results of Conversion on screen
Go to Option Routine

Option Routine
If another go, then go back to Preparation Length Conversion Routine
If not another go, then go back to main Control Routine

CHAPTER 5
The coding of the program

THE ENTIRE PROGRAM

This chapter will give you the entire program that has resulted from the rough design as discussed extensively in the previous chapter. The program will be presented here, routine by routine. Each part will be discussed, on the basis of the six points remaining to be explained at the beginning of Chapter 4. Where applicable you will be presented with flowcharts and/or program flows in words. This will emphasise heavily the top-down principles of structured programming.

GENERAL OBSERVATIONS

The program has been abundantly commented with REMark statements, whereas sections and modules have been separated by lines of dots, asterisks, hyphens, etc. This is to enhance the idea of MODULAR programming (programs consisting of procedure blocks, sub-programs, subroutines, or whatever you want to call them).

REM statements which explain the purpose of foregoing 'executable' statements, have been placed far right of the program listing (what you see here is the true program as printed out by a printer). REM statements start, in general, in the centre of a program line. When maintained throughout the program, this will provide a 'pretty print' listing, thus enhancing the look of the program and therefore its readability.

Readability is enhanced as well by the application of empty lines. Just type a line number and place a colon

(:) after it. An empty line can also be provided by a single REM statement.

At strategic points in the program you will see the following statement
PRINT" ⊐"
This is one of the Commodore screen functions. It will clear the screen. If you want to refresh your memory about this, then study the chapter about screen functions in your C64 manual.

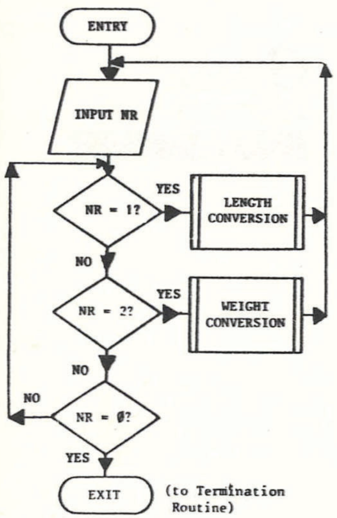
There are REMs in this program which can rightfully be considered as superfluous. They have been entered however to make the program as comprehensible as possible to the newcomer in the field.

THE MAINLINE COMES FIRST

(See Chapter 4).

'It is essential within the framework of top-down programming to start with the highest level routine, and consequently work on the next levels.' The mainline is the first level routine, and as we have established, will consist of three modules.

You will fully develop this mainline and ensure that it works, before you move on to the next level modules. Now let us consider each module of the mainline.



The Main Control Routine

This is the truly important first level routine, as this one must direct the program flow to the proper processing routines. This is how it has been set up (PROGRAM FLOW IN WORDS):

Enter Main Control Routine
Place Input prompts on screen
(1) for Length Conversions
(2) for Weight Conversions
(0) for Stop Program
If INPUT equals 1 then go to Preparation Length Conversion Routine, and after processing return to Main Control Routine
If INPUT equals 2 then go to Preparation Weight Conversion Routine, and after processing return to Main Control Routine
If INPUT equals 0 then go to Termination Routine
(if none of the above numbers, go back to first conditional test)


```
570 :
580 REM************ MAIN CONTROL ROUTINE *********************
590 :
600 PRINT                                   :REM PRINT BLANK LINE
610 :
620 PRINT"    *** MENU - MAKE YOUR CHOICE ***"
630 PRINT"    _______________________________"
640 PRINT
650 PRINT"        (TYPE IN APPROPRIATE NUMBER"
660 PRINT"          AND PRESS RETURN-KEY)"
670 PRINT
680 PRINT"          LENGTH CONVERSIONS (1)"
690 PRINT
700 PRINT"          WEIGHT CONVERSIONS (2)"
710 PRINT
720 PRINT"          STOP PROGRAM       (0)"
730 PRINT
740 PRINT"           (TYPE NR OF CHOICE)"
750 PRINT
760 INPUT"                 ";NR
770 :
790 IF NR = 1 THEN 960   :REM GO TO LENGTH CONVERSION ROUTINE
800 IF NR = 2 THEN 1130  :REM GO TO WEIGHT CONVERSION ROUTINE
810 IF NR = 0 THEN 870   :REM GO TO TERMINATION OF PROGRAM
820 GOTO 760             :REM IF NONE OF THE ABOVE NUMBERS
830 :                     REM THEN RETURN TO LINE 760
```

So what happens in this Main Control Routine is this. When the Input prompts have been placed on the screen (see lines 620—760) the program waits at line 760 for the INPUT of a number. It is not for nothing that a wide space has been placed between INPUT and the NR variable. This ensures that the question mark and the flashing cursor, prompting the user to type in a number, are placed nicely in the centre under the line (type number of choice).

Line 790 contains the first conditional test. IF NR = 1 THEN 960, which apparently is the beginning of the Preparation Length Conversion Routine. However, realise that when you code this part of the program, you will not know as yet where this particular routine begins. To make sure that the Main Control Routine works, just write
790 IF NR = 1 THEN 30000
and consequently something like
30000 PRINT "PREP.L.CONV.ROUTINE": STOP

What will happen now is that when you RUN the mainline, it won't "crash" in line 790 with an UNDEFINED STATEMENT ERROR, because the program goes somewhere! Do the same with line 800
800 IF NR = 2 THEN 40000.

Later on when you have finalised the second level routines in question, you will alter lines 790 and 800, so that indeed they will direct the program flow to the proper routines. (And erase lines 30000 and 40000.)

Line 820 contains a precautionary measure. It happens quite often that users accidently type in a wrong number. The line:

```
820 GOTO 760
```

anticipates this. After all, with any other number than 0, 1 and 2, all three tests will fail, and the program flow moves on and would end up in the termination routine. The flowchart will make that clear.

```
840 :
850 REM********** TERMINATION ROUTINE **********************
860 :
870 PRINT"□"                                :REM CLEAR SCREEN
880 PRINT"          END OF PROGRAM"
890 END                                 :REM END OF THIS PROGRAM
900 :
910 REM=============================================================
920 REM=============================================================
```

THE TERMINATION ROUTINE

This one speaks for itself. It clears the screen and then displays the message "END OF PROGRAM", and consequently ends the program with the statement END.

You must realise that, as stated before, in a case like this the Termination Routine is indeed very simple. Depending on the complexity of programs, however, Termination Routines can become complex as well. They can contain, for example, important output routines, or they may initialise the LOADING of other programs which you may have on your program tape or disk. But that is something for later.


```
100 PRINT"□"                                :REM CLEAR SCREEN
110 :
120 REM......... MAINLINE ..................................
125 :
130 REM********** INITIALISATION ROUTINE **********************
131 :
132 REM (IDENTIFICATION SECTION)
133 :
140 PRINT"      ****************************
150 PRINT"      *                          *
160 PRINT"      * CONVERSIONS IMPERIAL TO *
170 PRINT"      *  METRIC AND VICE VERSA   *
180 PRINT"      *                          *
190 PRINT"      ****************************
200 :
210 REM AUTHOR:              EDWARD P. BARTON
220 REM DATE WRITTEN:        JANUARY 5   1984
230 REM FINAL VERSION
240 :
250 REM PROGRAM DESCRIPTION:THIS PROGRAM CONVERTS STANDARDS
    OF MEASUREMENTS
260 REM AND WEIGHTS OF THE IMPERIAL SYSTEM AND THE METRIC SYSTEM
270 :
```

THE INITIALISATION ROUTINE

(See Chapter 4.) 'Though it is not a truly essential part of the program, it is always to handy to initialise a program with an identification section. For the sake of clear documentation it is also important to include a section which describes the variables used in the program'.

An Initialisation Routine normally contains a very essential part, as already hinted at (see Chapter 4). It may contain the definition of constant variables, definition of functions, and the dimensioning of arrays (again, more about that in a later chapter). You won't find such a section in this Initialisation Routine, because the entire program is in effect so simple that such special measures were not necessary.

```
280 REM-------------------------------------------------------------
285 :
290 REM (DESCRIPTION OF VARIABLES)
300 :
310 REM  NR IS NUMBER ENABLING MAINLINE TO BRANCH OUT TO 2ND
    LEVEL MODULES
320 REM  A$ AND B$ ARE DUMMY (STRING-)VARIABLES
330 REM  KG CONTAINS KILOGRAMMES
340 REM  LB CONTAINS POUNDS
350 REM  MT CONTAINS METRES
```

```
360 REM  FT CONTAINS FEET
370 REM  GR CONTAINS GRAMMES
380 REM  DG CONTAINS DECAGRAMMES
390 REM  HG CONTAINS HECTOGRAMMES
400 REM  TN CONTAINS METRIC TONS
410 REM  OU CONTAINS OUNCES
420 REM  SN CONTAINS STONES
430 REM  QU CONTAINS QUARTERS
440 REM  CW CONTAINS HUNDREDWEIGHTS
450 REM  IT CONTAINS IMPERIAL TONS
460 REM  MM CONTAINS MILLIMETRES
470 REM  CM CONTAINS CENTIMETRES
480 REM  DM CONTAINS DECIMETRES
490 REM  DA CONTAINS DECAMETRES
500 REM  HM CONTAINS HECTOMETRES
510 REM  KM CONTAINS KILOMETRES
520 REM  IN CONTAINS INCHES
530 REM  YA CONTAINS YARDS
540 REM  ML CONTAINS MILES
550 REM  KN CONTAINS KNOTS
555 :
560 REM=======================================================
```

As for the Identification Section and the section which describes the variables, the writer of this program started the coding of the ENTIRE job with the Identification Section, and FINISHED it with the description of the variables (This pertains to point 9 — see Chapter 4 — which says: 'Postpone refinements to the final stages of the programs'.) After all, it is not always accurately known beforehand how many variables are going to be used. Therefore the programmer reserved considerable memory space between lines 30 and 580 (the starting line of the Main Control Routine) and filled them out afterwards with the REM statements which describe every variable.

```
930 :
940 REM********** PREPARATION LENGTH CONVERSION ROUTINE *****
950 :
960 PRINT"□"                          :REM CLEAR SCREEN
970 :
980 PRINT:PRINT"       (LENGTH CONVERSIONS)"
990 :
1000 PRINT                :REM PRINT BLANK LINE
1010 :
1020 GOSUB 3070           :REM GENERAL SCREEN MESSAGE SUBROUTINE
1030 :
1040 INPUT"          ";A$
1050 IF A$ = "M" THEN 1680:REM GO TO INIT. CONV. METRIC-IMPERIAL
1060 IF A$ = "I" THEN 1860:REM GO TO INIT. CONV. IMPERIAL-METRIC
1070 GOTO 1040           :REM IF NOT M OR I THEN BACK TO 1040
1080 :
1090 REM-------------------------------------------------------
```

```
1100 :
1110 REM********** PREPARATION WEIGHT CONVERSION ROUTINE ******
1120 :
1130 PRINT"□"                          :REM CLEAR SCREEN
1140 :
1150 PRINT:PRINT"      (WEIGHT CONVERSIONS)"
1160 :
1170 PRINT                :REM PRINT BLANK LINE
1180 :
1190 GOSUB 3070           :REM GENERAL SCREEN MESSAGE SUBROUTINE
1200 :
1210 INPUT"          ";A$
1220 IF A$ = "M" THEN 1320:REM GO TO INITIALISATION M-I (WEIGHT)
1230 IF A$ = "I" THEN 1500:REM GO TO INITIALISATION I-M (WEIGHT)
1240 GOTO 1210           :REM IF NOT M OR I THEN BACK TO 1210
1250 :
1260 REM=======================================================
```

The Preparation Length Conversion Routine and the Preparation Weight Conversion Routine

TWO ROUTINES

Both routines are on the second level, and are basically the same. They follow a similar Input and decision structure as the Main Control Routine. There is, however, a subroutine involved which is used by both modules. It is Subroutine 3070 and it is referred to in lines 1020 and 1190.

It is a very simple subroutine, with one purpose only — avoiding double work. In both cases there is a screen message required which prompts the user to type in either an I or an M if he/she wants to have conversions from respectively Imperial to Metric, or Metric to Imperial. Subroutine 3070 provides that screen message. You will find this subroutine at the end of the program listing.

For clarity here is the program flow in words:

Clear screen

Display title of this Routine (Length Conversions)
Display input prompts "METRIC TO IMPERIAL, TYPE M & RETURN KEY"
"IMPERIAL TO METRIC, TYPE I & RETURN KEY"
If INPUT equals M go to Initialisation Metric to Imperial
If INPUT equals I go to Initialisation Imperial to Metric
If INPUT not equal to M or I then go back to first test

NOTE: See here the first deviation from the general plan, which states that from this module the program would branch to the Output module. Well, it appeared that some extra modules were required. As told in the previous chapter this deviation from the original plan has been brought in deliberately. But it must be repeated that the overall idea has been maintained as you will see.

```
1270 :
1280 REM********** INITIALISATION METRIC TO IMPERIAL WEIGHTS ****
1290 :
1300 REM (STANDARD INPUT UNIT IS THE KILOGRAMME)
1310 :
1320 PRINT"□"                          :REM CLEAR SCREEN
1330 :
1340 PRINT"TYPE NR OF KG & HIT RETURN-KEY"
1350 PRINT                             :REM PRINT BLANK LINE
1360 INPUT"          ";KG
1370 :
1380 PRINT"□"                          :REM CLEAR SCREEN
1390 PRINT KG;:
1400 :
1410 PRINT "KILOGRAMMES EQUAL:"
1420 GOTO 2060                  :REM GO TO CALCULATION SECTION
1430 :
1440 REM-------------------------------------------------------
1450 :
1460 REM********** INITIALISATION IMPERIAL TO METRIC WEIGHTS ****
1470 :
1480 REM (STANDARD UNIT INPUT IS POUND (LB))
1490 :
1500 PRINT"□"                          :REM CLEAR SCREEN
1510 :
1520 PRINT"TYPE NR OF POUNDS & HIT RETURN-KEY"
1525 :
1530 PRINT                      :REM PRINT BLANK LINE
1540 INPUT"          ";LB
1550 KG = LB * 0.4536           :REM CONVERT POUNDS INTO KILOGRAMMES
1560 :
1570 PRINT"□"                   :REM CLEAR SCREEN
1580 PRINT LB;:
1590 PRINT "POUNDS EQUAL:"
1600 GOTO 2060                  :REM GO TO CALCULATION SECTION
1610 :
1620 REM=======================================================
```



```
1630 :
1640 REM********** INITIALISATION METRIC TO IMPERIAL (LENGTH) ****
1650 :
1660 REM (STANDARD INPUT IN METRES)
1670 :
1680 PRINT"□"                          :REM CLEAR SCREEN
1690 :
1700 PRINT"TYPE IN NR OF METRES & HIT RETURN-KEY"
1705 :
1710 PRINT                      :REM PRINT BLANK LINE
1720 INPUT"          ";MT
1730 :
1740 PRINT"□"                   :REM CLEAR SCREEN
1750 :
1760 PRINT MT;:
1770 PRINT "METRES EQUAL:"
1780 GOTO 2450                  :REM GO TO LENGHT CALCULATION ROUTINE
1790 :
1800 REM-------------------------------------------------------
1810 :
1820 REM********** INITIALISATION IMPERIAL TO METRIC (LENGTH) ****
1830 :
1840 REM (STANDARD INPUT IN FEET)
1850 :
1860 PRINT"□"                          :REM CLEAR SCREEN
1870 :
1880 PRINT"TYPE IN NR OF FEET & HIT RETURN-KEY"
1885 :
1890 PRINT                      :REM PRINT BLANK LINE
1900 INPUT"          ";FT
1910 CM = FT * 30.48            :REM CONVERT FEET TO CENTIMETRES
1920 MT = CM / 100              :REM CONVERT CENTIMETRES TO METRES
1930 :
1940 PRINT"□"                   :REM CLEAR SCREEN
1950 :
1960 PRINT FT;:
1970 PRINT "FEET EQUAL:"
1980 GOTO 2450                  :REM GO TO CALCULATION ROUTINE LENGTHS
1990 :
2000 REM=======================================================
```

FOUR THIRD-LEVEL ROUTINES

We now have four third-level routines:
Initialisation Metric to Imperial Weights
Initialisation Imperial to Metric Weights
Initialisation Metric to Imperial Length
Initialisation Imperial to Metric Length

They all have one thing in common: an Input is required of a number which is to be converted. The first of the four gives the screen message "TYPE NR OF KG & HIT RETURN KEY". The typed in number will be stored in variable KG, then PRINTs (in line 1390) after clearing the screen (Clear Screen command in

line 1380) of the contents of this variable, followed by the words "Kilogrammes Equal". Note that line 1390 does take care of this. PRINT KG (semi-colon, colon) places the PRINT statement "KILOGRAMMES EQUAL" after the displayed contents of KG. After this the program proceeds to the Calculation section of Weights, which is at the same time the Output Routine.

Take note of the fact that all four third-level routines follow the same principles as regards the program flow.

But take special note of the fact that in two of these routines (Imperial to Metric Weights, and Imperial to Metric Lengths) one very basic conversion is executed — Pounds are converted into Kilogrammes, and Feet are converted into Metres.

THE CRUCIAL PART: NUMBER CRUNCHING

It is here where the actual task of the program is about to begin — the number crunching which must produce the CORRECT output. At this stage you, the programmer, are truly required to think logically and try to find the most efficient solution you can think of, at the moment (do not be surprised to discover a better solution when the program — or any program for that matter — has been completed; it is one of the reasons why so many programs are updated afterwards).

Realise that your computer is in fact a dumb machine. When you write the wrong code, it will produce the wrong result (remember we then talk about "logic errors"). So the general advice is to think carefully. Make sure that the calculations will be done correctly. When you have written the program PLAY COMPUTER with it, before you even enter it into the machine. Playing computer just means that you hand check a complete chain of calculations to find out whether the program will produce the right answers to the arithmetic. A hand held calculator is very helpful on crucial points like this, by the way.

But how has the problem been solved here? The aim of the program is to produce an output with conversions in both systems, Metric and Imperial, be it length or weight. That means that an input in Metres must be converted not only to the Imperial standards, but also to the other Metric standards. The same applies for input in Feet.

Not only will it be converted into Metric standards but also into the other standards of the Imperial system. This means that both conversions can have the same output format, which simplifies the matter considerably. We can go even further in our simplification of the problem. All conversions can be derived from only a few basic conversions. Just look at the following figure:

1 Metre
- = 1000 Millimetres
- = 100 CENTIMETRES
- = 3.2808399 Feet
- = 0.1 Decametre
- = 0.01 Hectometre
- = 0.001 Kilometre

1 Foot
- = 30.48 CENTIMETRES or 0.3048 Metre
- = 12 Inches
- = 1/3 Yard
- = 1/(3 * 1760) Miles (1 mile = 1760 yards)
- = 1/(3 * 2025) Nautical (1 mile = 2025 yards)

Note the connection Metre-Feet, and look again at the total figure. It amounts to one thing — WHEN ONE VALUE IS KNOWN, YOU KNOW THEM ALL. On this basis you can write a calculation routine which calculates on a step by step basis all other values, derived from the one value you put in. This principle has been maintained in both streams — conversion Length and conversion Weight.

Study these sections carefully. I strongly suggest you do some arithmetic of your own, to figure out precisely what has been going on here.

```
2010 :
2020 REM********** CALCULATION ROUTINE + OUTPUT OF WEIGHTS **
2030 :
2040 REM (METRIC UNITS WEIGHTS)
2050 :
2060 GR = KG * 1000          :REM CALCULATION OF GRAMMES
2070 DG = KG *  100          :REM CALCULATION OF DECAGRAMMES
2080 HG = KG *   10          :REM CALCULATION OF HECTOGRAMMES
2090 KG = KG *    1          :REM CALCULATION OF KILOGRAMMES
2100 TN = KG / 1000          :REM CALCULATION OF METRIC TONS
2110 :
2120 REM (IMPERIAL UNITS WEIGHTS)
2130 :
2140 LB = KG / 0.4536        :REM CALCULATION OF LBS
2150 OU = LB * 16            :REM CALCULATION OF OUNCES
2160 SN = LB / 14            :REM CALCULATION OF STONES
2170 QU = LB / 28            :REM CALCULATION OF QUARTERS
2180 CW = LB / 112           :REM CALCULATION OF HUNDREDWEIGHTS
2190 IT = LB / 2240          :REM CALCULATION OF IMPERIAL TONS
2200 :
2210 REM (OUTPUT OF RESULTS - OF WEIGHTS)
2220 :
2230 PRINT                               :REM PRINT BLANK LINE
2240 PRINT"       GRAMMES "; GR
2250 PRINT"   DECAGRAMMES "; DG
2260 PRINT"  HECTOGRAMMES "; HG
2270 PRINT"   KILOGRAMMES "; KG
2280 PRINT"   METRIC TONS "; TN
2290 PRINT                               :REM PRINT BLANK LINE
2300 PRINT"        OUNCES "; OU
2310 PRINT"        POUNDS "; LB
2320 PRINT"        STONES "; SN
2330 PRINT"      QUARTERS "; QU
2340 PRINT"HUNDREDWEIGHTS "; CW
2350 PRINT" IMPERIAL TONS "; IT
2360 :
2370 GOTO 2820               :REM  GO TO OPTION ROUTINE WEIGHTS
2380 :
2390 REM------------------------------------------------------
```

THE OUTPUT ROUTINES

As you see both Output routines are in actual fact also the calculation routines. There is not much to say about it, because they are both very straightforward. One little remark though. See line 2160 with the variable SN, which contains the weight in Stones. Perhaps you may have thought that ST would be a better choice as a variable to indicate weight in Stones.

As far as Commodore computers are concerned NEVER use ST, because it is a RESERVED VARIABLE. The machine needs ST for purposes other than storing any kind of number. The C64 manuals will tell you more about this.


```
2400 :
2410 REM********** CALCULATION ROUTINE + OUTPUT OF LENGTHS *******
2420 :
2430 REM (METRIC UNITS LENGTHS)
2440 :
2450 MM = MT * 1000          :REM CALCULATION OF MILLIMETRES
2460 CM = MT *  100          :REM CALCULATION OF CENTIMETRES
2470 DM = MT *   10          :REM CALCULATION OF DECIMETRES
2480 MT = MT *    1          :REM CALCULATION OF METRES
2490 DA = MT /   10          :REM CALCULATION OF DECAMETRES
2500 HM = MT /  100          :REM CALCULATION OF HECTOMETRES
2510 KM = MT / 1000          :REM CALCULATION OF KILOMETRES
2520 :
2530 REM (IMPERIAL UNITS LENGTH)
2540 :
2550 IN = CM / 2.54          :REM CALCULATION OF INCHES
2560 FT = IN / 12            :REM CALCULATIONS OF FEET
2570 YA = FT / 3             :REM CALCULATIONS OF YARDS
2580 ML = YA / 1760          :REM CALCULATIONS OF MILES
2590 KN = YA / 2025          :REM CALCULATION OF KNOTS
2600 :
2610 REM (OUTPUT OF RESULTS IMPERIAL LENGTHS)
2620 :
2630 PRINT                           :REM PRINT BLANK LINE
2640 PRINT"MILLIMETRES "; MM
2650 PRINT"CENTIMETRES "; CM
2660 PRINT" DECIMETRES "; DM
2670 PRINT"     METRES "; MT
2680 PRINT" DECAMETRES "; DA
2690 PRINT"HECTOMETRES "; HM
2700 PRINT" KILOMETRES "; KM
2710 PRINT                           :REM PRINT BLANK LINE
2720 PRINT"     INCHES "; IN
2730 PRINT"       FEET "; FT
2740 PRINT"      YARDS "; YA
2750 PRINT"      MILES "; ML
2760 PRINT"      KNOTS "; KN
2770 :
2780 GOTO 2940               :REM GO TO OPTION ROUTINE LENGTH
2790 :
2800 REM=============================================================
2810 :
2820 REM********** OPTION (WEIGHT) ROUTINE **********************
2830 :
2840 PRINT:PRINT              :REM PRINT TWO BLANK LINES
2850 :
2860 PRINT"MORE? TYPE Y OR N, THEN RETURN-KEY";:
2870 INPUT B$
2880 IF B$ = "Y" THEN 1130  :REM RETURN TO INIT. WEIGHT CONVERSION
2890 IF B$ = "N" THEN 3200  :REM BACK TO MAIN CONTROL
2900 GOTO 2870              :REM IF NOT N OR Y RETURN TO 2870
2910 :
2920 REM-------------------------------------------------------
2930 :
2940 REM********** OPTION (LENGTH) ROUTINE **********************
2950 :
2960 PRINT:PRINT                      :REM PRINT TWO BLANK LINES
2970 :
2980 PRINT"MORE? TYPE Y OR N, THEN RETURN-KEY";:
2990 INPUT B$
```

```
3000 IF B$ = "Y" THEN  960 :REM RETURN TO INIT.LENGTH CONVERSION
3010 IF B$ = "N" THEN 3200 :REM BACK TO MAIN CONTROL
3020 GOTO 2990             :REM IF NOT Y OR N THEN BACK TO 2990
3030 :
3040 REM============================================================
3050 :
3060 :
3070 REM******** GENERAL SCREEN MESSAGE SUBROUTINE **************
3080 :
3090 PRINT"METRIC TO IMPERIAL, TYPE M & RETURN-KEY"
3100 PRINT"IMPERIAL TO METRIC, TYPE I & RETURN-KEY"
3110 PRINT:PRINT           :REM PRINT TWO BLANK LINES
3120 RETURN
3130 :
3140 REM******** RETURN TO MAINLINE *******************************
3150 :
3200 PRINT"□".                  :REM CLEAR SCREEN
3210 GOTO 600                   :REM RETURN TO MAIN CONTROL
```

THE OPTION ROUTINES

We have already indicated in our general flowchart of the entire program (and also in the written program flow), that the user must be offered a choice when the program RUN only. That is why I now bring in 'no', because finishing a program in this crude way is not nature (Length or Weight)? If the answer is yes then the program flow will return to the entry point of that conversion stream. The Initialisation Routine Length or Weight, depending on what you are in at the moment. If not, i.e. you either want to try out conversion stream, the Initialisation Routine Length or Weight, depending on what you are in at the return to the mainline — in this case the Main Control Routine. These two Option Routines (both on the fifth level) speak for themselves.

NOTE that both Option Routines have ONE EXIT POINT in the very small Return to Mainline routine. It clears the screen, and then, when the program flow re-enters the Mainline, the main menu is displayed on the screen, and you can start over again with conversions, just by hitting the 1 or 2. Or if you want to finish, hit the 0.

As for the two very small screen message subroutines, they speak for themselves.

IT IS YOUR TURN

You have just studied a program which has been written entirely in accordance with the rules of structured programming and the top-down approach. The result is a program which, just by looking at it, is easy to understand.

But now it is your turn. Type in this program on a module by module basis, and test it every time a module has been entered. It will happen then that the program stops (the computer will then display an appropriate error message, but don't be disturbed by that). But you can use the technique as described under the mainline. Where a module "exits" to another module that hasn't been written yet, write a "dummy" program line in its place. For example, see lines 1210 to 1240, in the Preparation Weight Conversion Routine. In these lines the program is supposed to branch out to another program module but when such a module isn't there, then write line 1220 something like this:

```
1220 IF A$ = "M" THEN 60000
```

Line 60000 can then be used as a "dummy program line" simulating modules which, in actual fact, haven't been developed as yet. Line 60000 may look like this

```
60000 PRINT "MODULE NOT YET AVAILABLE": END
```

This dummy program line, or you may even call it a dummy program module, can be utilised every time you test a program where following modules aren't developed as yet.

BUGS

When you do not have much experience in typing (but also when you ARE a good typist!) you will make typing errors, causing the program to "crash". The computer will then deliver error messages on the screen.

As pointed out, usually they are due to typing errors — you may have forgotten to type a colon, or you may

type a colon where a semi-colon was required. You may forget quotation marks, you may type IPUT instead of INPUT and so on. Take your time to figure out these errors and correct them.

By the way, computer programmers call an error a "bug". Correcting errors is generally known as "debugging".

WHAT NEXT?

Let us assume that you typed in the entire program and made it run flawlessly. You may then ask 'What next?'. Well that is up to you. You are now ready to be on your own. From now on you must rely on your own inventiveness and creativity.

Realise that your tool for programming — your computer — is an extremely patient device. It will never become bored, it will never become angry when you do something wrong. There is also very little chance that you can destroy your machine.

The only thing that can happen is that it "locks up", which simply means that some keystrokes may cause it to go into an endless loop within the electronic structure of the machine. But do not panic then — just switch off your machine and switch it on again. It will be back to normal. The fact that the program has been developed on the basis of the top-down approach makes it possible for you to EXTEND the program. It is relatively easy to add some extra modules, which for example enable conversions such as Metric to Imperial Volumes and vice versa. Naturally you can also add a temperature conversion module.

The important thing to do then is add a few extra prompts and tests in the Main Control Routine. For example:

Volume Conversions (3)
Temperature Conversions (4)

You then add another two conditional tests in the lines 790 to 820.

CHAPTER 6
Other tools of BASIC

REVISION

In the previous chapters you have become acquainted with the following tools of the BASIC computer language: INPUT, PRINT, Variables, arithmetic operators, interpunction functions (.,:;'"'), REM, IF-THEN, GOTO, GOSUB-RETURN.

With this rather limited set of BASIC instructions you have already written a simple but correctly functioning program. In this and the next chapter we will discuss many more BASIC commands, statements, functions, etc. And now you have become more familiar with programming and programmers' jargon, I will introduce you to some more terminology.

STRING AND INTEGER VARIABLES

You have already come across some string variables. See as an example lines 1050 and 1060 in the conversion program (chapter 5).

```
1050 IF A$ = "M" THEN 1680
1060 IF A$ = "I" THEN 1860
```

Note the dollar sign after the A, and note above all the quotation marks around the letters M and I. When a variable has a dollar sign added to it, it becomes a string variable. (This is also called an "alpha numeric variable". A string variable can contain both ALPHAbetic characters and NUMERIC characters.) However, such variables can only store data WHICH HAVE BEEN PLACED BETWEEN QUOTATION MARKS.

Examples:

A$ = "JOHNNY"
X$ = "AB-12"
NM$ = "LONDON"
NR$ = "123456789"

When you forget to place the data between quotation marks, this will happen. Type A$ = 1 and hit RETURN. The computer responds with ?TYPE MISMATCH ERROR, because there is a mismatch between the type of variable (it is a string variable), and the data presented to it (here a number, not placed between quotation marks).

IMPORTANT

Commodore BASIC accepts variable names which are longer than two characters. A variable name may be sixteen characters long. However, the computer recognizes the first two characters only! The rest is ignored.

EXAMPLE

Type JA = 10 and hit RETURN. Now type PRINT JANET and hit RETURN again. The computer responds with 10. However, when you type PRINT JAN, hit RETURN, or PRINT JAQUELINE, hit RETURN, or any other variable name which begins with JA, and the computer will in all cases respond, or in computer jargon — RETURN, with 10, as it recognizes only the two first letters of the variable name. In the example the first two letters of all names were J and A.

This leads to the advice NOT to use variable names longer than two letters, as the chance is too great that what appears as entirely different variable names to you, is not at all different for the computer. Variable names longer than two letters are only useful when the programmer makes absolutely sure that in all cases the first two letters of a variable name are not the same in any other variable name.

Variable names in a case like this will function correctly:

DATE . . . . . DAte
MONTH . . . . . MOnth
YEAR . . . . . YEar

because the first two letters of every variable name are entirely different when compared to the other.

INTEGER

Here I have to explain the notion "integer", if you are not aware of its meaning. A variable can store numbers, as you know. You must distinguish between "real" and "integer" numbers. A real number has digits before and after the decimal point. The digits before the decimal point form the INTEGER, the digits after the decimal point form the FRACTION. Hence a number like this 12.3456789 is a real number, 12 is its integer, 0.3456789 is its fraction.

An integer variable contains only the integer of a number, and is specified by the placement of the percent symbol (%) after the variable name. Examples: A%, AB%, XX%. Now type into your computer A% = 55.6677 and hit RETURN. Then PRINT A% and hit RETURN. The computer displays 55 instead of 55.6677. A% is an integer variable and therefore stores only the integer of 55.6677, which is 55.

NOTES

There are three variable names which you are not allowed to use as normal variables — ST, TI and TI$. More information about those three is supplied by your computer manual.

A variable of whatever type ALWAYS begins with a letter, never with a digit or any other symbol. The combination 1A, for example, is not accepted by the computer as a variable. A1, A2, A3, A4 etc. are accepted.

AND, OR, NOT

These three words are called "logical operators", and they are to be used in combination with IF-THEN statements. As you know, the IF-THEN combination makes for a conditional test. When you incorporate the "logical operators" AND, OR, NOT you can do more than one test in one go.

Examples:

IF A = 5 AND B = 10 AND C = 15 THEN D = 20

This program statement can only be executed when all three conditions are met:

If A = 5 AND B = 10, **BUT C = 18,** then the overall condition is not met and the program line is ignored.

IF A = 3 OR A = 7 OR A = 13 THEN END

In this case the line will be executed when A contains one of the three numbers (3, 7 or 13).

Try this one;

```
10 INPUT A,B,C
20 IF NOT A=3 AND B=7 OR C=8 THEN 40
30 GOTO 10
40 PRINT "THIS IS SILLY"
50 END
```

Puzzle about its how and why.

With this knowledge in mind you can now examine and type in this very simple guessing game:

```
10 PRINT "TYPE IN ANY NUMBER AND HIT RETURN"
20 CN = 1 : REM COUNTER OF GUESSES
30 INPUT A: REM TYPE IN ANY NUMBER
40 IF A = 45 OR A = 72 OR A = 93 THEN 100
50 CN = CN +1
60 GOTO 30
100 PRINT
110 PRINT "YOU ARE CORRECT"
120 PRINT
130 PRINT "IT TOOK YOU";CN;" GUESSES TO FIND"
140 PRINT "ONE OF THE THREE POSSIBLE NUMBERS"
150 END
```

Naturally, this game is very simple indeed because you already know the three possible numbers. But this does not matter. I just demonstrated a principle. It will be more interesting when you replace line 30 by 30 INPUT A,B and line 40 by 40 IF A = 45 AND B = 67 THEN 100. You can also insert an extra line, 45 PRINT" SORRY, YOU ARE WRONG".

THE + SYMBOL AS 'STRING OPERATOR'

Just as with the equals symbol (=) the plus symbol (+) has a dual function (= can mean equals, but it also means 'replace by'). Not only is + the arithmetic operator which adds A to B, but it can also "concatenate" string variables. In other words, by adding one string to the other it can make up a new string. Example:

```
10 A$ = "MON"
20 B$ = "DAY"
30 C$ = A$ + B$
40 PRINT C$
```

The result will of course be MONDAY. By the way PRINT (A$ + B$) will give the same result.

DATA AND READ

DATA is a so-called non-executable statement. You could call it an indicator instead, because it indicates where there is data available to be processed. A DATA statement is something like this:

10 DATA 5.7, 8.3, 1000, 45, "LONDON", "ASD", 10 .22,0,0,999

Each data item is separated by a comma from the previous one. Note also that in this line of data you see numerics and strings. It is permissible to mix numeric data and strings but it is not recommended. Lines with mixed data types will easily lead to program errors.

Now what is the function of all this? Data are there to be READ. DATA lines like the example are senseless when there is no READ statement somewhere in the program. The following example will clarify what the DATA READ combination is up to. Type in this program and RUN it.

```
10 READ A
20 IF A > 25 THEN PRINT A
30 IF A = 999 THEN 50
40 GOTO 10
50 END
60 DATA 60,15,22,40,68,24,31,44,666,721,999
```

As you will discover this program prints all numbers in the DATA row which are greater than 25. The program is a loop, and with every cycle one item of the series of data is READ, one after the other. You will also see that the DATA line has been placed at the bottom of the program. It is said that you can place DATA virtually everywhere in a program, but it is good programming practice to do so either at the end of the program or at the beginning, in the initialisation routine. A collection of DATA items can be called the DATA pool.

Again a little exercise (see previous program). Erase line 30 (type 30 and hit RETURN). Now RUN the program. You will see how the program again prints all numbers which are greater than 25. However, the program stops with an error message: ?OUT OF DATA ERROR IN 10. This is because you made an endless loop of this program by removing line 30. There is no way now to branch out of this loop. The program does stop, however, because after reading eleven items of data, it has run out of data and then the program "crashes".

There are some rules to remember when you use the DATA READ combination.

(1) You can use as many DATA items in a program as you may need, but do realize that a list of data items can only be as long as a program line will allow. Hence put no more than 79 characters on a line, including the line number and the DATA statement. When you have more DATA, just start a new line, (line-nr, DATA, item,item, etc).

(2) When DATA is to be READ, make sure that the variable into which an item must be stored is of the same type. Hence numbers are to be placed into numeric variables (there is an exception to that, see later on), and strings MUST be moved into string variables. Thus the following list of DATA items:

10 DATA 10, 12, 15, 7, 90, 16, 122, 9999

are to be READ and moved into variables of the type A, AA, XY, C1, H5, and so on. A list of DATA items:

20 DATA "LONDON","NEW
YORK","SYDNEY","SINGAPORE"

can only be READ and placed into variables of the type A$, AA$, XY$, C1$, H5$, and so on.

NOTE: As a string variable is an alphaNUMERIC variable as well, it will accept numbers in an alphanumeric DATA list. However, the computer will then treat them as if they are strings. This has the consequence that, initially, you can't do arithmetic with, for example, the number 3 placed in a string variable. More about this later, when we examine the VAL statement.

NOTE: strings can also be placed in DATA lists without quotation marks. Hence 20 DATA LONDON, NEW YORK, SYDNEY, SINGAPORE will do the job perfectly. A last remark. When the computer encounters two or more commas in a DATA list, it will assume that the space between the commas is occupied by a zero or an empty string. Hence a DATA statement 30 DATA 15,,,,56,72,,55 is perfectly valid. The computer reads it as 30 DATA 15,0,0,0,56,72,0,55.

RESTORE

This statement is used in combination with the READ and DATA statements. When a list of DATA items which already have been READ must be used again, they can't be READ anymore, unless they are RESTOREd. Go back to our, already altered, example program, and replace line 50 END by 50 RESTORE:GOTO 10. RUN this program. With dazzling speed the same set of numbers (all higher than 25) repeats itself over and over again. The effect of RESTORE is that the DATA 'pointer' of the computer is reset and points to the first item of the entire list of DATA items. By entire list, I mean ALL items available in a program.

DEF FN

This statement stands for "define function" and is one of the most powerful programming tools Commodore BASIC can offer, and particularly so in the area of rather complicated calculation procedures. It can in fact replace in one single program line an entire subroutine, and it can provide extra functions.

Take, for example the function INT, which stands for INTeger. With this function, the computer will display — or "return" as programmers prefer to say — the integer of a real number. Just type:

X = INT(45.6789) and hit RETURN
PRINT X and hit RETURN

The computer will return 45, the integer of the number 45.6789.

However, there are enough situations where the user requires the fraction of a real number. This could be obtained by the following:

X = 45.6789 and hit RETURN
X = X − INT(X) and hit RETURN

The computer will return .6789 which is the fraction of the number.

In principle the line X = X − INT(X) could be treated as a subroutine, to be placed somewhere at the end of the program. This, however, is not always practical because everytime you want to find the fraction of a variable, this variable has to be replaced by X, then the program flow goes to the subroutine, returns, and the variable X has to be changed again. This is rather cumbersome, and can be avoided by using a DEF FN statement. The format of the DEF FN statement is as follows:

DEF FN name (dummy variable) = formula with the dummy variable

"Name" means that the function must have a name, preferably one which relates to the formula it wants to solve. The name "dummy variable" just means that you can take any variable, because when the new function is used in the program, this 'dummy' will be replaced by any variable used for the solving of the problem in question. The function which will return a fraction of a real number can therefore be:

10 DEF FN FR(X) = X − INT(X)

A little program will demonstrate this.

Type in (NEW and hit RETURN first when a previous program is still in your machine):

```
10 DEF FN FR(X) = X - INT(X)
20 INPUT A      : REM INPUT A REAL NUMBER
30 A = FN FR(A) : REM CALCULATION OF FRACTION
40 PRINT A      : REM PRINTS FRACTION
50 END
```


Now run this program and when the computer asks you to input a real number, ENTER for example the number 123.4567890. The computer will return, after you have entered this number, and hit the RETURN key, with 0.45678890, which is the fraction (though not entirely, as the computer has some problems with precision in this area).

Now note two things in this little program.

(1) The define function has been placed at the beginning of the program. THIS IS MANDATORY. Before any program which has defined functions can run properly, those functions really have to be defined before they can be applied. Obviously, this can only be done at the very beginning of the program, i.e. is in the Initialisation Routine.

(2) See line 30. The dummy variable X of FN FR(X) has been replaced by the variable A. This is to make clear to you that every variable, placed within the framework of a DEF FN statement, will undergo the treatment as set out in the function. Hence when you want to find the fraction of any other variable just do this:

B = FN FR(C), or B = FN FR(K), or B = FN FR(B).

When properly worked out beforehand a DEFine FuNction statement can replace entire subroutines.

DIM

DIM stands for DIMension, and again this statement is a very powerful programming tool, but unfortunately it is also one of the least understood BASIC statements.

For this I have to bring in the notions of "subscripted variables" and "ARRAYS". Those two notions are very closely related as you will see. The idea of subscripted variables is that they have something in common. To illustrate this, let us go back to the example of a filing cabinet (see chapter 2), and pull the drawer (or Variable) A. Imagine that in this drawer A is hanging an ARRAY of files, each containing information (data) which have something in common, namely that they are all placed in drawer A (variable A). You could give each of those hangers a name, something like A(1), A(2), A(3), and so on.

Now, this is exactly what happens in BASIC. An ARRAY of data can be stored in an array of subscripted variables. The format of a subscripted variable is: A(digit), or A%(digit), or A$(digit). Hence all three normal variable types can be subscripted (numeric real, numeric integer, alphanumeric (string)).

NOTE: do realize that a subscripted variable A(1) is NOT the same as A1!

What has the DIM statement to do with all of this? Everything. DIM is crucial here, because it will determine how many data an array of subscripted variables can handle. The array has to be DIMensioned first! Let me enlarge on that. When you switch on your computer, it automatically assumes TEN subscripts available for any subscripted variable. (Keep in mind that counting to ten means here from 0 (zero) to 9, not 10! A computer accepts the 0 (zero) as a totally valid number). Hence there is no problem when you want to place data in all subscripted variables from A(0) to A(9). Thus you are not doing anything wrong when you type A(3) = 3, or A(9) = 999, or A(0) = 11111.

But there will be problems when you want to place data into a variable with a subscript higher than 9. Just try this:

Type A(12) = 45 and hit the RETURN key.

Your computer will respond with the message ?BAD SUBSCRIPT ERROR. This is because you have not DIMensioned this subscript variable in the first place! To demonstrate what I mean, try this:

Type DIM A(20) and hit RETURN
Type A(15) = 92 and hit RETURN
Type PRINT A(15) and hit RETURN

The computer returns 92. There are no problems any more because the subscripted variable A() has been properly DIMensioned. But again, when you exceed the dimension of 20 in this case, the computer will respond with the ?BAD SUBSCRIPT ERROR message.

Now there are some rules pertaining to the proper use of the DIM statement.

DIM AND ITS RULES

(1) It is proper programming practice to place DIM statements at the very beginning of the program, that is in the Initialisation Routine. (The general rule is, place DIM statements before the FIRST executable statement).

(2) The format of the DIM statement is DIM (nr). The number (nr) determines the size of the array. Hence DIM A(100) means that in the array A(100), 101 items can be stored.

(3) One DIM statement can dimension several arrays. Example:

```
10 DIM A(20), B(100), C$(15), FA%(70)
```

NOTE: the comma which separates each dimensioned variable.

There is nothing wrong with DIM A(20): DIM C$(15): DIM DA%(70), but of course that is rather impractical. The first rule is very important. A subscripted variable can be dimensioned ONLY ONCE. If the computer encounters a DIM statement for a second time, that is with a variable which already has been subscripted, the program will come to a halt and the computer displays the message ?REDIM'ARRAY ERROR.

To demonstrate this, just type in this program:

```
10 INPUT A
20 DIM A(20)
30 A(B) = A
40 PRINT A(B)
50 B = B + 1
60 GOTO 10
```

RUN this program and see what happens when you have entered a number for the second time. Right . . . ?REDIM'ERROR IN 20. The program should be:

```
5 DIM A(20)
10 INPUT A
30 A(B) = A
40 PRINT A(B)
50 B = B + 1
60 GOTO 10
```

The program will work correctly now, though there is still the limitation of the array size which you made twenty. But note the program does not encounter the DIM statement anymore after it has been initialized at the very beginning of the program! Therefore, DIM statements must be placed, as already said, in the Initialisation Routine of any program.

NOTE: in BASIC an array can be DIMensioned up to a size of 255. Hence you can store up to 255 items of data into an array.

Be aware of line 30 in the example program. The subscript here is B. This is allowed of course, as the computer considers a variable to be a number anyway. Note also that the subscript can contain a real number, that is a number with an integer, a decimal point and a fraction. But in that case the computer will only recognize the integer. Hence a subscript variable like A(C) where C = 1.8 is seen by the computer as A(1).

What is so useful about DIM and subscripted variables? They are very useful, because they give you the opportunity to store data which are of the same nature in an array of the same nature. For example, you can store the names of major capital cities into string variables like this:

AM$ = "AMSTERDAM", LO$ = "LONDON",
PA$ = "PARIS", RO$ = "ROME", and so on.

However, to retrieve all these names makes for rather lengthy programming. It will be a different matter with a string array, for example CC$(). CC$(1) can contain Amsterdam, CC$(2) can contain London, CC$(3) Paris, and so on. Later a simple program routine will enable the user to retrieve all those names in one go, when he uses the subscript variable CC$(X), where X contains a VARYING number. Look at the following routine:

```
100 X = 1
110 PRINT CC$(X)
120 IF X = 10 THEN 150
130 X = X + 1
140 GOTO 110
150 END
```

This routine will print ten capital cities (provided of course that ten capital cities are stored in the CC$ array). It is for this reason that arrays are widely used in BASIC programming. They save lots of memory space, because they shorten routines, and make programming very efficient.

There are a few things you still must know about DIM. So far we have discussed the DIM statement in this form only — DIM A(20). What the programmer has done in such a case is define a "one dimensional array". However, the statement DIM can also be used to define two dimensional arrays, and even three dimensional ones. In such cases the statement is written DIM A(20,5), which is two dimensional, or DIM A(20, 5, 10) which is three dimensional. This is something I advise you not to worry about at this stage of your programming career. Multi dimensional arrays are for advanced programming techniques, and therefore not within the scope of this book.

END STOP CONT

We discuss these commands in one go, because they are closely related. You have already encountered END — it does not need any extra explanation. Except that END is optional, i.e. in certain cases it is not necessary to use it. In some simple, very straightforward programs the last line is indeed the last line. Without the END statement the program ends there anyway. But when the program must terminate somewhere else in the program, for example at the end of the mainline, then the END command is mandatory.

If a program stops at an END statement, you canNOT restart the program by typing CONT and then RETURN — this contrasts with what the manual says. The STOP command is of a different calibre. A STOP command can be placed almost anywhere in a program, just to let the program come to a halt. The computer will then display the message BREAK IN (a line number follows) and READY. If you want to continue then type CONT and hit the RETURN key.

The STOP command can be very useful when you are in the process of writing a program. It gives you the opportunity to examine intermediate results.

CHAPTER 7

Still more tools of BASIC

FOR-TO-NEXT-STEP

Here you will be introduced to a set of highly essential BASIC statements, as they enable you to construct program loops. In previous chapters we have already discussed some looping techniques and exercises with the GOTO and IF statements.

The combination FOR-TO-NEXT-STEP however, makes writing of BASIC programs which utilise loops a great joy, as it enhances your versatility enormously.

As the manual says, 'This statement works with the NEXT statement to repeat a section of the program A SET NUMBER of times'.

The format (or syntax) is:

FOR (name of variable) × (start nr of count) TO (end nr of count) STEP (loop variable, that is count by).

An example:

```
10 FOR I = 1 TO 100 STEP 2
```

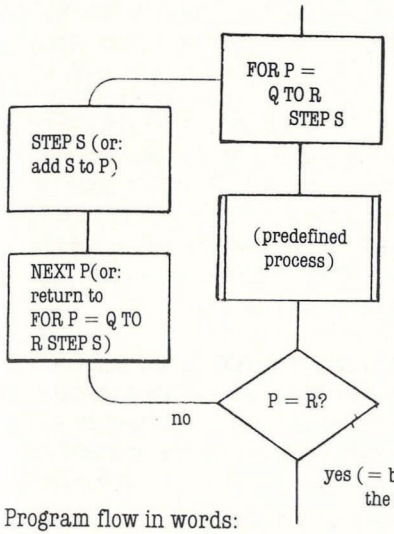
Let us first discuss:

```
10 FOR I = 1 TO 100
20 PRINT I;
30 NEXT I
```

Type in this program and RUN it.

What happens is that the program prints horizontally the numbers 1 to 100 on the screen and then stops. How does it work?

In line 10 the programmer determines that the contents of variable I will be altered in increasing sequence a hundred times. Line 20 prints the present contents of variable I. Line 30 EXAMINES the present contents of variable I AND IF THE CONTENTS OF I IS NOT 100 the contents of I will be increased by the value 1 and the program-flow returns to line 10 which has the beginning of the chain FOR-TO-NEXT. This goes on until the moment that the test in line 30 reveals that the number 100 has been placed in I. In that case the program flow won't return to the beginning of the chain but will move to the next line after 30. For a proper understanding of this programming tool study the next flowchart thoroughly:



GENERAL FLOWCHART OF FOR-TO-NEXT-STEP CHAIN

Note: STEP S can be left out; in that case the computer assumes the value of S equals 1.

Hence FOR P = Q TO R

Program flow in words:

Set up FOR-NEXT loop: variable equals Q (begin nr) to R (end number)
Execute processing (predefined process)
Test P: If P equals R then branch out of FOR-NEXT loop. If P does not equal R then go back to beginning of loop, and add loop variable S to P.
Restart cycle.

The best way to illustrate a FOR-NEXT loop is a little program. Type in:

```
10 DIM A(20)  :REM DIMENSION SUBSCRIPT VARIABLE A()
20 DATA1,5,12,3,0,23,55,46,24,34,22,7,9,33,29,94,4,8,10,11
30 FOR I = 1 TO 20
40 READ A(I)
50 NEXT I
60 REM THIS IS THE FIRST FOR-NEXT LOOP - NOW ANOTHER ONE
70 FOR J = 1 TO 20
80 PRINT A(J)
90 NEXT J
100 END
```

RUN this program and see what happens. Indeed, it displays all the numbers of the DATA list of line 20, thanks to the two FOR-NEXT loops.

Actually, the first FOR-NEXT loop comprises the lines 30, 40, and 50. Line 30 says that the variable "I" will be increased by one unit every cycle, for a total of 20 cycles that this loop will run (FOR I equals 1 to 20!). So in the first cycle I=1. In line 40 the program pointer encounters the first subscript variable, which will be effectively A(I) or A(1). The statement now READs the first DATA item and places it in A(1). This first DATA item happens to be 1 as well (see the list in line 20). The program goes to line 50, which says NEXT I. In this line a test takes place, I=20? If not, the value of 1 is added to I, and the program counter jumps back to line 30, and goes through the entire procedure of lines 30, 40 and 50 again. When after 20 cycles (after all, FOR I=1 TO 20) the value of I has increased to 21, then the built-in conditional test of line 50 (NEXT I) proves to be true, and the program branches out of the loop.

In this way, i.e. with a FOR-NEXT loop, an entire list of DATA items can be READ into an array. Hence in this case, the subscripted variables A(1) through A(20) are filled with these items of data, one at a time.

After the program branches out of the loop at line 50, it goes to line 70 (60 is ignored as it contains a REM statement) where it again encounters a FOR-NEXT loop. The purpose of this one is to print on the screen the contents of every subscripted variable of the A() array. Now the variable to be increased and tested is J, and the value is ascending from 1 to 20, just as with I. So this cycle starts at line 70, prints the value of subscripted variable A(J), increases the value of J by one unit in line 90 and the program printer goes back to line 70. The cycle ends when J becomes 21. All values of the array are thus displayed on the screen.

A little exercise. Replace line 60 with:

60 FOR J = 1 TO 20 STEP 2

and RUN again this program. Now you will see only ten numbers appear on the screen. What is more,

there is a number taken from the DATA list, the next number however is missing, then again a number, the next number is missing, and so on.

In this way only eleven uneven subscripts of the array are to be assessed. The computer displays the contents of A(1), A(3), A(5), and so on. As you will understand, there is a STEP OF 2 UNITS between two subscripted variables.

IMPORTANT: the FOR-NEXT loop can also work the other way around, that is with a DECREASING loop variable. Replace line 60 with:

60 FOR J = 20 TO 1 STEP −1

RUN the program again and now note how the program displays firstly the contents of A(20), then A(19), then A(18), and so on. NOTE: STEP −1 is mandatory here, when you want to decrease the loop variable with one unit per cycle. As you know, when the loop variable is increasing by one per cycle the STEP statement is optional, not so the other way around.

When you are fully in the business of programming, you will appreciate this particular feature of BASIC more than anything else, as it does so many useful jobs such as highly repetitive procedures, which is precisely the prime task every computer has been designed for!

I must point out that this feature becomes particularly useful when you are going to write so called "nested loops". I advise you, however, to postpone this application until you have advanced to a reasonably high level of programming skill. Nested loops, which are described in virtually every book on programming in BASIC, are a rather tricky business, hence not within the scope of this book.

GET

This command truly does what it says. GET something. What? A character from the keyboard. To demonstrate this see the next program:

```
10 GET A$
20 PRINT" HELP !"
30 PRINT" I AM CAUGHT IN A VICIOUS CIRCLE"
40 PRINT" SAVE ME BY HITTING A KEY"
50 PRINT" SORRY..., DON'T KNOW WHICH ONE.."
```

```
60 PRINT
70 IF A$ = "G" THEN 100
80 GOTO 10
100 PRINT
110 PRINT" THANKS SO MUCH...PFFF, I AM
    EXHAUSTED.."
120 END
```

RUN this program. By the way, it is almost impossible to read the text, so fast does it scroll over the screen. This can be remedied just be pressing the CRTL key (on the left side of the keyboard) and the program slows down considerably.

If you now hit the G on the keyboard, the program will come to a halt, after it has branched out of line 70. The GET command takes in the G character, tests it in line 70, finds the condition to be true, and therefore branches out to line 100. If the condition is found to be false — with every character other than G the program loops back, in line 80, to line 10, and goes on for ever and ever, unless you hit the G key.

The GET command is extremely useful in the setting up of user friendly screen menus. To demonstrate this, go back to our big conversion program. See lines 760 to 820.

Replace these lines by the following:

```
760 GET NR$: IF NR$ = "" THEN 760
770 :
790 IF NR$ = "1" THEN 960
800 IF NR$ = "2" THEN 1130
810 IF NR$ = "0" THEN 870
820 GOTO 760
```

RUN this altered program. The menu appears on the screen but there won't be a question mark and a flashing cursor, however. But, when you now hit one of the prompts (1, 2 or 0), the program reacts immediately, you do not have to hit the RETURN key first!

What happens in the (replaced) lines 760 is the following. In line 760 the computer expects a character from the keyboard. As long as there is no character from the keyboard the NR string (NR$) contains NOTHING (then NR$ = ""). In that case line 760 forms an endless loop, which can be interrupted only when the tested character (keyed in on the keyboard) is 1, 2 or 0 (zero).

Now you have experienced the power of the GET command, you can similarly alter the remaining prompts in the conversion program. See lines 1040-1060, 1210-1230, 2870-2890, 2990-3020, and 3155-3170. Try it out, it will be a rewarding exercise.

THE ON STATEMENT

This statement can be very handy. It must be used in combination either with GOSUB or GOTO. The format of this statement is:

ON X GOTO
ON X GOSUB

It depends on the value of X as to where the program flow will go. To demonstrate this, type in the following program and RUN it.

```
10 INPUT "DAY-NR  "; DA
20 INPUT "MONTH-NR"; MO
30 INPUT "YEAR-NR "; YE
40 ON MO GOSUB 100,110,120,130,140,150,160,170,180,190,200,210
50 PRINT
60 PRINT
70 PRINT "THE DATE IS ";DA;MO$;YE
80 END
100 MO$ = "JANUARY  ":RETURN
110 MO$ = "FEBRUARY ":RETURN
120 MO$ = "MARCH    ":RETURN
130 MO$ = "APRIL    ":RETURN
140 MO$ = "MAY      ":RETURN
150 MO$ = "JUNE     ":RETURN
160 MO$ = "JULY     ":RETURN
170 MO$ = "AUGUST   ":RETURN
180 MO$ = "SEPTEMBER":RETURN
190 MO$ = "OCTOBER  ":RETURN
200 MO$ = "NOVEMBER ":RETURN
210 MO$ = "DECEMBER ":RETURN
```

How does this work? Very simple. In the program there is line 30 which asks for an input of the number of the month. In line 40 this number is evaluated against the NUMBER OF SUBROUTINES offered here. When the number stored in MO equals 1 then the program flow is directed to the FIRST subroutine in line 100. When the number stored in MO equals 2, the program flow is directed to the second subroutine in line 110. When the number stored in MO equals 7, the program flow is directed to the seventh subroutine line 160, and so on. The same principles as outlined here with the GOSUB command, apply to the GOTO command.

IMPORTANT: as you see ON is a two letter command, and can therefore easily be mistaken for a two letter variable. It is not. When you try

```
5 ON = 10
```

the computer will reply with a syntax error.

OPEN CLOSE CMD

These commands are all related to the communication between the computer and 'peripheral devices': printer, disk drives, cassette recorders. We will devote an entire appendix to these matters, and hence refer you accordingly.

MANIPULATION OF STRINGS

Page 128 of your manual shows you some "string functions", the explanation of which leaves much to be desired. String manipulation is an advanced programming technique, but most of the string functions are easily understood, which is why I include them in this chapter.

ASC(X$), ASC("A") and CHR$(X) are functions which belong to the realm of very advanced programming, and we won't discuss them. Save this for a later date with another book.

LEFT$, RIGHT$, MID$

These functions enable you to pick parts of a string. Recapitualate the properties of strings. Strings are stored in string variables, a variable name followed by a dollar sign. A string may consist both of numeric and alphabetic characters, provided they are enclosed in quotation marks. Something like:

```
X$ = "1234567890 ABCDEFGHIJK"
Y$ = "JAQUELINE VANDERBILT"
Z$ = "======================"
```

are all correct applications of strings. By the way, the last one shows computer graphics, which officially do not belong to the realm of alphanumerics, but are accepted by the computer nonetheless.

Again, string functions enable you to manipulate data contained in a string. Here is an example to demonstrate this point. There are several ways to input a date (day, month, year) into a program. You have encountered one method already in a previous chapter.

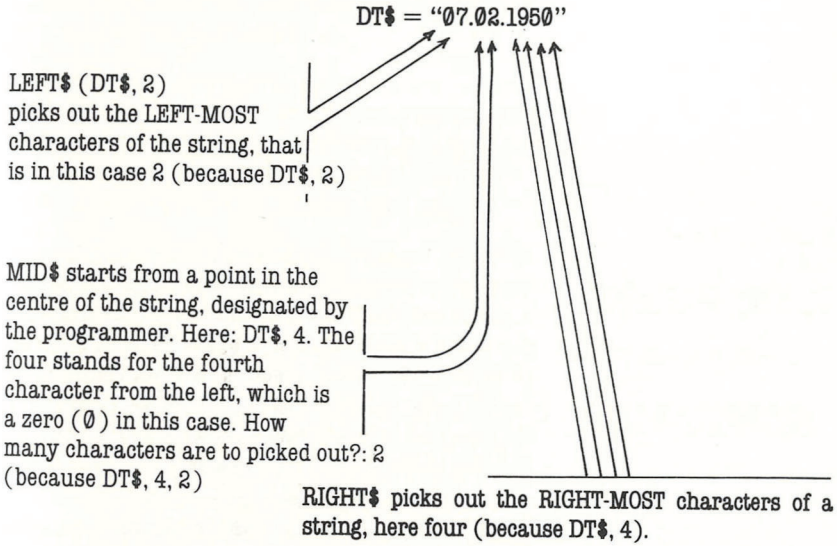
Another way to do this can be something like:

```
10 INPUT "DATE (MM.DD.YYYY)";DT$
```

This means that, for example, a date of 7 February 1950, must be input in the format DD.MM.YYYY which results in 07.02.1950. Hence the string variable DT$ will contain "07.02.1950". How do you pick out the different items in this string? See the following:

```
20 DA$ = LEFT$ (DT$, 2)
30 MO$ = MID$ DT$, 4, 2)
40 YE$ = RIGHT$ (DT$, 4)
50 PRINT
60 PRINT DA$, MO$, YE$
```

Type in this program (also line 10) and RUN it. Indeed, it shows the date now split up into its different items. How did that happen? Here again is the date:



You will appreciate these particular features of BASIC when you are a little further advanced in programming. It truly enables you to play around with words, parts of words, and so on.

STR$(X)

This function turns a numeric into an alphanumeric, or string. It is sometimes easier to handle a string numeric than just a plain numeric, specifically so when you have to print figures in tables (rows and colomns). The idea is simply this:

```
10 X = 123
20  X$ = STR$(X)
30 PRINT X$
```

This program will turn the value in X into a string value. There are situations where this can become very useful, as pointed out already, for example when you want to print very neatly in a column. In BASIC plain numerics are often hard to manipulate. For example, if you want to have the following numbers printed in a column:

```
   123.456
    23.500
 10225.009
```

do not think that you can accomplish this by

```
10 X = 123.456 : Y = 23.500 : Z = 10225.009
20 PRINT X
30 PRINT Y
40 PRINT Z
```

This is what you get:

```
123.456
23.5
10225.009
```

However, when you turn those numbers into strings, then you can write your program in such a way that X turns into X$, Y into Y$, Z into Z$. In the first case you add the following strings to X$: X$ = "bb" + X$ (this adds two empty spaces to the left side), Y$ = "bbb" + Y$ + "00" (this adds three empty spaces to the left, and two zero's to the right). NOTE: the "b" indicates a BLANK space to be typed in.

Then, when you print these altered strings, you get a perfect column:

```
   123.456
    23.500
 10225.009
```


NOTE: you should know that turning a number into a string by the STR$(X) function has one little drawback. It always adds one empty space on the left side. Hence X$ = STR$(123.456) gives " 123.456" (note the blank after the first quotation mark. Sometimes this extra space is a nuisance, and in that case a RIGHT$(X$,X) function may bring the correct result, because it enables you to erase that unwanted blank. See:

```
Z$ = " 10225.009"
Z$ = RIGHT$(Z$,9)
```

Z$ then becomes "10225.009"

VAL(X$)

When numbers are placed in a string variable, they will become useless for arithmetical manipulations. For example, when A$ contains 10 and B$ contains 20 you can't do a thing like this

```
C$ = A$ + B$
```

and then expect that C$ will contain 30. On the contrary, C$ may contain "10 20" or "1020" (the two strings have been "concatenated" here). Multiplication, division, and substration however, are totally out of the question.

But it will be a different matter when you use the VAL(X$) function. This function establishes the numeric value of the string. So when you do C = VAL(A$)+VAL(B$), then you will get the value 30 in C. The manual gives you, on page 129, some extra examples.

SPREAD YOUR WINGS

As you may have noticed, we have left quite a few, mainly numeric, functions untouched. If you are fond of mathematics however, those numeric functions will already have some meaning to you, and you won't have much trouble in finding out how they work. In a separate appendix you will find useful predefined functions (created with the DEF FN statement) which will make it easier for you to do trigonometric calculations and the like.

This book has given you a head start in BASIC programming. When you have done all the exercises, small and large, you will have acquired such skills that you do not need more explanations. You are able to find out for yourself. You are ready now to spread your wings, so my advice is, fly and explore the world of BASIC programming on your own.

As for your Commodore 64, you know that it can do a lot more than plain arithmetic, manipulation of text, etc. The C 64 offers you "sprites" (which help you set up animation) and colour and sound (enabling you to make a musical instrument of your Commodore 64). A final piece of advice — do not jump too hastily into these areas. Become proficient in the basics of BASIC first, before you endeavour to tackle sound, graphics and colour. But when you are ready, the manuals of the Commodore 64 and the Commodore 64 Programmers Reference Guide offer you excellent guidance. I wish you lots of success, and fun.

APPENDIX A

How to handle "peripherals" (tape-recorder, disk-drive, printer)

When you bought your Commodore 64, you undoubtedly bought a little "datasette" cassette recorder as well. Or perhaps a disk drive instead, and even a printer maybe.

All these extra devices are known as "peripherals". Because they are used on the periphery of the computer. They are not part of the computer, but they can only work in combination with it. To make them work, you must not only connect them 'physically' (that is, connect them with cables) but also electronically. A peripheral, no matter what it is, will not work just like that. For example, do not think that a printer will print, just because your program contains the PRINT command.

On the contrary, the printer will not respond, because the computer has not been connected electronically to the printer, though there is the physical connection of the cables. The printer, and the other devices, will only respond when you incorporate some special BASIC commands into your programs.

This appendix gives you some insight into those special commands, which control the workings of peripherals. We will cover the following:

OPEN, CLOSE, CMD, PRINT#, LIST, LOAD, SAVE, VERIFY.

But first something about "files" and device numbers.

FILES

The word "file" is often used in computer jargon, but just as often it gives rise to confusion as regards to its precise meaning. One book says 'a group of organised data (records), assembled for one particular purpose and considered as one unit. It is stored in permanent

storage such as disks or tapes'. In that sense a program in itself is seen as a file, but so are sets of data (for example, records of personal data of employees). And indeed, when you study manuals on disk drives later on, they usually never refer to programs to be stored, but to files to be stored on disk. So a file can contain records or programs. Whatever the case, a file has to be OPENed when the computer needs access to it and it must be CLOSEd afterwards.

OPEN

This is an I/O statement (Input/Output) which will open a file related to a particular peripheral, such as the printer, the disk drive, or in particular cases the cassette recorder. The format of the statement is:

OPEN file number, device number, address, "file name, type, mode".

In most cases you do not need all these different codes. Only the two first ones (file number, device number) will do. It depends on the complexity of the operation. When you want the computer to make contact with a printer, the OPENing statement is usually OPEN 4,4. The first 4 is not mandatory — often another number will do. But there is general agreement amongst Commodore programmers to maintain OPEN 4,4. Hence the first 4 is the file number, and the second 4 is the device number.

As for disk drives, the OPEN 15, 8 statement is often used. Here 15 is the file number, and 8 the device number. As working with disks is somewhat more involved than working with printers or cassette recorders, the OPEN statement often becomes a little larger, OPEN 15, 8, 15. The last 15 is then the "secondary address" or "path". If you want to know more about this, see the Commodore 64's Programmers Reference Guide.

CLOSE

When a device has been connected electronically by the OPEN command, there comes a situation when you have to disconnect it (electronically). In that case you CLOSE the connection by closing the access

to the device. Hence when you execute CLOSE 4, you disconnect electronically the printer (and therefore also close file 4 or whatever), and when you CLOSE 8, you disconnect electronically device 8, which is the disk drive.

DEVICE NUMBERS

A computer is a unit in itself, and every extra device to which it is connected physically, and with which it is supposed to communicate, can be contacted by its own device number. Commodore assigns the following numbers to its respective devices: 1 for cassette recorder, 4 for printer, and 8 for disk drive.

HANDLING OF PRINTERS

CMD

This stands for Command. It will COMMAND a device to perform a certain task. In the case of a printer the format will be CMD 4. What it then does is to send data (considered to be output) to the printer, instead on the screen. For example, if you want to see output on the screen you type PRINT A, and the computer will display the contents of A on the screen. However, when you want this output on paper, instead of on the screen, you can type the following line: OPEN 4,4 CMD 4: PRINT A, at the appropriate position in your program.

A program set in this CMD 4 mode will not show any output on the screen, but all print statements will produce output on the printer's paper. NOTE: it must be said that this is not a very popular way to obtain "hard copy" (output on paper). The reason is that the CMD 4 statement makes it hard for the programmer to have a program alternate between screen output and printer output, which in many cases is desirable. To demonstrate this, type in this program (NOTE: it will will only work when you have a printer!):

```
10 OPEN4,4: CMD 4
20 PRINT "TYPE IN YOUR NAME"
30 INPUT"NAME";NM$
40 PRINT NM$
50 PRINT#4:CLOSE 4
60 END
```

Indeed, what you get is not precisely what you had in mind. The INPUT cursor blinks on the screen alright, but there is no prompt "NAME". And when you input your name it will be printed on paper, but at the left connected with the NAME prompt of line 30. This problem can be solved easily by the PRINT# command as explained in the next paragraph.

PRINT

It is the "hash" (#) symbol which solves the problem of the previous paragraph. Here is a re-written program:

```
10 OPEN4,4
20 PRINT "TYPE YOUR NAME"
30 INPUT "NAME"; NM$
40 PRINT#4, NM$: PRINT NM$
50 PRINT#4: CLOSE 4
60 END
```

WHEN you run this program you get an output both on screen and on paper.

With 40 PRINT NM$ you get output on the screen only.

With 40 PRINT#4, NM$ you get output on paper only.

As said, the second statement is much preferred when it comes to the production of printed output (on paper).

Important: You may have been aware of the fact that the PRINT command has an abbreviation — the questionmark (?). When you type ?A and hit RETURN the computer will print the contents of A on the screen. Later on, in the program listing you will find that the question mark has been converted to the full word PRINT. HOWEVER, THE PRINT# COMMAND CANNOT BE TYPED AS ?#! The computer will not accept this. Type it in full, always. Another thing to remember is the format if you want to have the contents of variables printed by this command, there MUST be a comma between the hash symbol (#) and the variable! As you have seen demonstrated in the above examples.

SUMMARY

To summarize the handling of commands which control the communications between computer and printer:

OPEN 4,4 opens the connection to the printer.

CMD 4 sends all output to the printer, also PRINT statements which are screen messages only.

PRINT works in combination with CMD 4, and prints everything, including screen messages, on paper.

PRINT#4, X only prints on paper what is meant to be printed on paper, when the connection to the printer has been OPENed (by 4,4) and the CMD 4 command has been omitted.

PRINT#4 MUST be used when the connection between computer has to be closed.

CLOSE 4 closes the connection — or in computer jargon, closes the file.

So here is the general format:

```
10 OPEN 4,4
20 PRINT#4, "MESSAGE, OUTPUT, ETC"
30 PRINT#4
40 CLOSE4
```

A CLOSE command can be placed virtually anywhere in a program when the peripheral is not needed anymore. It is good programming practice to CLOSE "files" or peripherals in the Termination routine of a program.

LIST

This command is very useful when you want a LISTing of your program or parts of the program, ON PAPER. The format of the sequence of commands is this:

OPEN 4,4: CMD 4: LIST
and hit RETURN

The program is then printed out in its entirety. However, you can also have printouts (listings) of just the lines, or routines, you want. Then type LIST linenumber — linenumber. One line only — LIST linenumber. For example if you want to have the

program listed between lines 100 and 200, just type LIST 100-200 and hit RETURN. If you want line 100 only, type LIST 100 and hit RETURN. When the LISTing has been completed, type PRINT#4: CLOSE 4 and hit RETURN.

NOTE: The LIST command used on its own lists the program on the screen.

TAPE RECORDER AND DISK DRIVE HANDLING

The commands SAVE, LOAD and VERIFY are to be used with the cassette tape recorder and the disk drive. It must be asserted that the Commodore 64 manual is clear enough about LOADing and SAVEing of programs to and from cassette tapes. I advise you to study and gain some practical experience from pages 18 — 22 of the manual, if you haven't done so already. Pages 110 — 111 detail a very useful application as well; creating data files on tape.

However, where this manual and also the Programmers Reference Guide leaves you very much in the cold, is disk drive handling. The manual that comes with the disk drive is, at the time of this writing, not much better. The newcomer in the field will be prone to much confusion when he/she tries to decipher the contents of the disk drive manual. Therefore I attempt to bring some clarity in this matter here.

When you take a blank disk out of the box for the first time, you CANNOT use it straight away. The diskette is truly blank, the disk drive and the computer cannot do very much with it. Therefore the first step is as follows.

INITIALIZE THE DISK

Put the disk into the disk drive (of course, it has been connected and switched on), close the door, and then type into your computer:

OPEN 15,8,15,"NEW0: disk name, Id"
or OPEN 15,8,15,"N0: disk name, Id"
and hit the RETURN key.

This is what it means:

OPEN opens a file, numbered 15, to device number 8 (the disk drive, remember) using a path, which is numbered 15.

NEW0 (New ZERO) tells the computer and the disk drive that you have a new disk in the drive. When there is no more than one drive connected to the computer, the number of the drive is zero (hence NEW0). Realize, it is drive 0 but DEVICE 8.

The colon (:) after N0 is mandatory.

The disk must have a name, which can be any, provided it is not longer than sixteen letters.

The comma after disk name is mandatory.

Id stands for Identity, and can be any two character combination, for example X1, A1, CX etc.

The quotation marks are mandatory.

When you execute this chain of commands the disk drive will run for a while (a whirring sound, and a little light is on), and then stop. You can then take out the disk which is now ready for use.

SAVING PROGRAMS ON DISK

Now you have initialised your disk you can SAVE programs on it. This is essentially the same process as saving programs onto tape. However, when you are allowed with tapes to type SAVE"PROGRAM-NAME" and hit RETURN, in the case of a disk you MUST do it the following way:

SAVE"program name", 8

and hit RETURN. The comma and the 8 are mandatory.

When you have more than one disk drive, and you save the program on a disk which is placed in a drive other than drive 0, you must type:

SAVE"drive-number:program name",8.

So when the number of the other drive is 1, it will be

SAVE"1: program name",8.

That is all there is to it.

IMPORTANT

In virtually no Commodore disk drive manuals is it made clear that you can very easily save more than one program on a disk. Perhaps it has been overlooked because it is so simple. When a program is already on a disk, it cannot easily be wiped off or overriden by another program as is the case with cassette tape. When you want to save another program on a disk, just follow the normal saving procedure. Automatically, the new program will be placed after the first program on the disk, the next one comes next, and so on.

SAVING PROGRAMS IN STAGES OF DEVELOPMENT

When you are in the course of developing a program, this is not a matter of hours, but often of days, weeks and months. Naturally you want to save the parts of the programs you have worked out. After all, when you switch off the computer, the program is lost. There is no problem in saving these programs when you use a cassette tape.

However, when you use a disk, then every time that you want to save what you have developed so far, you MUST re-INITIALIZE the disk, before you save the latest version of your program. See the paragraph about Initializing disks. If you do not initialize the disk again, you will have several versions of your program on your disk, before you know it. And unless you have given each version a distinctively different name, it will be hard for the disk drive to select the correct one.

VERIFYING SAVED PROGRAMS

It is good practice to VERIFY a program just saved on tape or disk. In the case of a tape, you have to rewind the tape, then type VERIFY and hit RETURN. In the case of a program just saved on disk, you type:
VERIFY " program name",8 and hit RETURN

In the case of a different drive number than 0 (1 for example):
VERIFY "1: program name", 8 hit RETURN

LOADING PROGRAMS FROM DISK

When there is only one program on the disk, then:
LOAD "*", 8 and hit RETURN.
When there is more than one program on the disk:
LOAD" program name", 8
In the case of a different drive number than 0 (for example 1):
LOAD"1: program name", 8 and hit RETURN

HANDLING USED DISKS

No problems here. When you want to use a disk which already contains programs, which, to your opinion are obsolete, you can clear the entire disk by re-initializing it. You can also delete programs of your choice, by using the SCRATCH command. See:
OPEN 15,8,15,"S0: name program or file"

The S stands for Scratch.

DISK DIRECTORY

How can you know how many programs there are on a disk, and if there is room left for other programs (to be recorded)? Simply type:

LOAD"$",8 and hit RETURN.

NOTE: when the disk is placed in drive other than 0, for example drive 1, then type:

LOAD"$1",8 and hit RETURN

When the computer signals READY, type LIST and hit RETURN again. The directory will be placed immediately on the screen — it shows the contents of the disk.

You can then select the program and LOAD it into the computer, and naturally you can establish how many blocks are free. A program takes so many blocks of

space on a disk. For example a 25 kilobytes program may occupy 98 or 99 blocks, and then still leave 566 or 565 blocks free, hence there is enough place left for many other large programs.

WARNING

Never leave a disk in a disk drive, when you have switched it off. Always remove it. When left in the drive, you may lose the program when you switch on the drive again. Reason, powering up the disk drive provides a sudden upsurge in magnetic fields, which then can destroy the contents of your disk.

PROTECT YOUR DISKS

Last warnings:

(1) Protect your programs on your disks by glueing a protection tab over the notch at the side of the protection cover. This prevents you from inadvertently wiping off programs.
NOTE: (something for later) disks are also used for storage of records. Unless such a disk is full, and files (records) must be written to it, a protection tab may not be placed over the notch.

(2) Handle your disks with care. Always keep them in their covers, keep them out of direct sunlight, store them in cool places, and keep them away from sources of electro-magnetism — electrical motors, loudspeakers, magnets.

APPENDIX B
How to handle the function keys (F1 — F8)

Both Commodore 64 manuals (the small manual and the Commodore 64's Programmers Reference Guide) mention the function keys only briefly or not at all, even though those keys are most conspicuously placed on the key board. Only the manual of the new portable Commodore 64, gives an expose.

You need the GET command to do the work (see the last chapter of this book). Every function key has its own CHR$ code (Character string code), which range from (133) to (140).

Here is the table of the relation between the function keys and their CHR$ codes:

f1 — CHR$(133) f5 — CHR$(135)
f2 — CHR$(137) f6 — CHR$(139)
f3 — CHR$(134) f7 — CHR$(136)
f4 — CHR$(138) f8 — CHR$(140)

To gain access to these keys, see the next program:

```
10 GET A$: IF A$ = "" THEN 10
20 IF  A$ = CHR$(133) THEN 100
30 IF  A$ = CHR$(134) THEN 200
40 IF  A$ = CHR$(135) THEN 300
```

In this way you can develop a program which enables direct access to different routines, just by hitting one of the function keys. By the way, realise that function keys 2, 4, 6 and 8 can only be accessed by pressing the SHIFT key first.

Imagine a screen menu which tells the user: HIT F1 FOR CONVERSION LENGTH METRIC TO IMPERIAL. Or: HIT F2 FOR CONVERSION LENGTH IMPERIAL TO METRIC. Or: HIT F8 FOR CONVERSION FAHRENHEIT TO CELCIUS.

APPENDIX C
Some useful mathematical functions in DEF FN format

TRIGONOMETRY

The Commodore 64 offers the trigonometric functions SIN, COS, TAN and ATAN. There are two problems connected to these functions.

(1) They work only on radians, and return results in radians. Many people prefer to work in degrees instead of radians.

(2) The inverse functions ARC SIN and ARC COS are not available. (ARC TAN is available as ATAN.)

To overcome these problems you can create some functions which deal with it adequately. To convert from radians to degrees, you have to use the famous constant PI. As you may have discovered, that constant is available on the key board of your Commodore 64. Thus when I refer to PI, you type the appropriate symbol instead.

SINE FUNCTION
```
DEF FN SI (X) = SIN (X*PI/180)
```

COSINE FUNCTION
```
DEF FN CO (X) = COS (X*PI/180)
```

TANGENT FUNCTION
```
DEF FN TA (X) = TAN (X*PI/180)
```

ARC SINE FUNCTION
```
DEF FN AS (X) = ATN (X/SQR (1-X*X)) * 180/PI
```

ARC COSINE FUNCTION
```
DEF FN AC (X) = (PI/2 - ATN(X/SQR(1-X*X))) *
180/PI
```

ARC TANGENT FUNCTION
```
DEF FN AT (X) = ATN(X) * 180/PI
```

How does this work: Example:


```
5 DEF FN SI (X) = SIN (X * π/180)
100 INPUT A
110 A = FN SI (A): REM FIND  SINE OF A
120 PRINT A
```

You input at line 100 in DECIMAL degrees, and line 120 will return the correct result. For example, when you put in 45, the result will be 0.707106781 which is correct.

NOTE: the input MUST be in decimal degrees, not in degrees, minutes and seconds. This appendix contains a defined function which converts degrees, minutes and seconds straight into decimal degrees.

INTEGER AND FRACTION

The Commodore 64 already has the built-in INT(X) function, which returns the integer of a real number. However, this is not always done correctly. For example, if you want the integer of a negative real number, you may get something which you do not expect. Try this:

```
A = −5.9: PRINT INT(A)
```
The computer returns −6 instead of −5.

This defined function will help:

```
DEF FN IN (X) = SGN(X) * INT(ABS(X))
```

Place this function at the beginning of a program and whenever you need the correct integer of a negative number, you will get it. Example:

```
10 DEF FN IN(X) = SGN(X) * INT(ABS(X))
20 INPUT A : REM INPUT A NEGATIVE REAL

30 A = FN IN(A)
40 PRINT A
```

When a program uses the integer of a real number, it may be handy to use the fraction as well. Here is a defined function which MUST be used in combination with the IN(teger) function:

```
DEF FN FR (X) = ABS(X) — FN IN (ABS(X))
```

HOURS (OR DEGREES) MINUTES AND SECONDS TO DECIMAL HOURS (OR DEGREES)

This function occupies an entire line, hence type it in carefully. No spaces are allowed between the characters!

```
DEFFNDE(X)=SGN(X)*(INT(ABS(X))+INT(100
*FNFR(ABS(X)))/60+FNFR(100*ABS(X))/36)
```

NOTE that the defined fraction has been used in this one. How to use it?

```
100 INPUT "HH.MMSS";HS
110 HS = FN DE(HS)
120 PRINT HS
```

NOTE the input format HH.MMSS. So if you want to convert, for example, 15 hrs 33 mins 17 secs, you type in 15.3317. When you convert it with the above formula the answer must be 15.5547222

Here is a function which converts decimal hours or degrees to Hours (or Degrees) and minutes only.

```
DEF FN SE (X) = FNIN (X) + FNFR(X) * 0.6
```

NOTE again that in this function, two other defined functions have been used. You must realize that in a case like this, those functions to be used within functions, MUST be placed in the program PRIOR to the functions wherein they will be used.

ROUNDING OFF FUNCTION

This function produces two digits after the decimal point, and rounds off as well.

```
DEF FN RD (X) = FNIN((X*100) + 0.5)/100
```

With this function a number like 45.67891, will be rounded off to 45.68.

IMPORTANT

As pointed out elsewhere in this book, DEF FN statements MUST be placed at the very beginning of the (executable) program, that is in the Initialisation Routine.

APPENDIX D

A Solution

A solution to the exercise in Chapter 2:

```
5 PRINT"□": REM CLEAR SCREEN
10 REM THIS PROGRAM CONVERTS POUNDS
15 REM STONES, OUNCES AND GRAINS
20 REM INTO GRAMMES
25 :
30 INPUT "ENTER NUMBER OF POUNDS";LB
32 :
35 REM CONVERT POUNDS INTO GRAMMES
40 GM = LB * 453.6
45 :
50 REM CONVERT STONES TO GRAMMES
55 SE = GM * 14
60 :
65 REM CONVERT OUNCES TO GRAMMES
70 OU = GM / 16
75 :
80 REM CONVERT GRAINS TO GRAMMES
85 GR = GM / 16 / 16
90 :
92 PRINT : REM PRINT BLANK LINE
95 PRINT "STONES TO GRAMMES :"; SE
100 PRINT"POUNDS TO GRAMMES :"; GM
110 PRINT"OUNCES TO GRAMMES :"; OU
120 PRINT"GRAINS TO GRAMMES :"; GR
130 :
135 END
```

APPENDIX E
Further Reading

LITERATURE

Commodore 64 Programmer's Reference Guide, Commodore Business Machines Inc. 1982

Commodore 64 Exposed, by Bruce Bayley, Melbourne House (Publishers) Ltd. 1983
This is a book for the more advanced Commodore 64 programmer.

Programming the CBM/PET by Raeto West, Level Limited, London, 1982
Though this book deals only with the forerunners of the Commodore 64, i.e. the CBM 2000, 3000 and 4000 series, much of it is applicable to the Commodore 64, as the Commodore 64 employs the same BASIC 2 version of Commodore BASIC.

BASIC With Style, Programming Proverbs, by Paul Nagin and Henry F. Ledgard, Hayden Book Company, Rochelle Park, New Jersey, 1978.
This is one of the very best books ever written on programming in BASIC. It deals extensively with the top-down approach of programming. This book is a must for every BASIC programmer who wants to learn the art, and wants to learn it very well.

The Working Commodore 64, A Library of Practical Subroutines and Programs, by David Lawrence, Sunshine Books.

This is just a selection from a wide variety of books available for the Commodore 64 computer. Many more can be browsed through in your specialised computer bookshop, and many more will be published.

INDEX

This book is designed for Commodore 64 users who want to begin writing their own programs. It assumes a working knowledge of the keyboard and the machine's basic functions. Ideal for anyone tired of typing in program listings, this gentle introduction to the jargon and complexities of computing will take the reader step by step through program design and structure. Illustrated throughout with clear examples, ***Discover Your Commodore 64*** makes full use of the machine's sound and graphic capabilities and prepares the user for advanced programming with a wealth of hints and tips.

CENTURY PUBLISHING CO. LTD.
ISBN 0 7126 0422 7

£2.95

DISCOVER YOUR COMMODORE 64 Philip Williams