beginner

COLLABORATORS

	TITLE : beginner	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		February 24, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

beginner

1.1 Introduction to Amiga E

```
Introduction to Amiga E
***********************
```

To interact with your Amiga you need to speak a language it understands. Luckily, there is a wide choice of such languages, each of which fits a particular need. For instance, BASIC (in most of its flavours) is simple and easy to learn, and so is ideal for beginners. Assembly, on the other hand, requires a lot of effort and is quite tedious, but can produce the fastest programs so is generally used by commercial programmers. These are two extremes and most businesses and colleges use C or Pascal/Modula-2, which try to strike a balance between simplicity and speed.

E programs look very much like Pascal or Modula-2 programs, but E is based more closely on C. Anyone familiar with these languages will easily learn E, only really needing to get to grips with E's unique features and those borrowed from other languages. This guide is aimed at people who haven't done much programming and may be too trivial for competent programmers, who should find the 'E Reference Manual' more than adequate (although some of the later sections offer different explanations to the 'Reference Manual', which may prove useful).

Part One (this part) goes through some of the basics of the E language and programming in general. Part Two delves deeper into E, covering the more complex topics and the unique features of E. Part Three goes through a few example programs, which are a bit longer than the examples in the other Parts. Finally, Part Four contains the Appendices, which is where you'll find some other, miscellaneous information.

```
  A Simple Program
```

1.2 A Simple Program

A Simple Program
================

If you're still reading you're probably desperate to do some programming in E but you don't know how to start. We'll therefore jump straight in the deep end with a small example. You'll need to know two things before we start: how to use a text editor and the Shell/CLI.

1.3 The code

The code

Enter the following lines of code into a text editor and save it as the file simple.e (taking care to copy each line accurately). (Just type the characters shown, and at the end of each line press the RETURN or ENTER key.)

```
PROC main()
  WriteF('My first program')
ENDPROC
```

Don't try to do anything different to the code, yet: the case of the letters in each word is significant and the funny characters are important. If you're a real beginner you might have difficulty finding the ' character. On my GB keyboard it's on the big key in the top left-hand corner directly below the ESC key. On a US and most European keyboards it's two to the right of the L key, next to the ; key. (If you don't have your keyboard set up properly then you find that keys don't produce the same characters that are printed on them--especially when use use the shift key. In this case it will probably behave like a US keyboard, although you should really fix this and set it up properly--see the manuals that came with your Amiga.)

1.4 Compilation

Compilation

Once the file is saved (preferably in the RAM disk, since it's only a small program), you can use the E compiler to turn it into an executable program. All you need is the file ec in your C: directory or somewhere else on your search path (advanced users note: we don't need the Emodules: assignment because we aren't using any modules). Assuming you have this

and you have a Shell/CLI running, enter the following at the prompt after changing directory to where you saved your new file:

```
ec simple
```

If all's well you should be greeted, briefly, by the E compiler. If anything went wrong then double-check the contents of the file simple.e, that your CLI is in the same directory as this file, and that the program ec is in your C: directory (or on your search path).

1.5 Execution

Execution

Once everything is working you can run your first program by entering the following at the CLI prompt:

```
simple
```

As a help here's the complete transcript of the whole compilation and execution process (the CLI prompt, below, is the bit of text beginning with 1. and ending in >):

```
1.System3.0:> cd ram:
1.Ram Disk:> ec simple
Amiga E Compiler/Assembler/Linker/PP v3.2e registered (c) '91-95 Wouter
lexical analysing ...
parsing and compiling ...
no errors
1.Ram Disk:> simple
My first program1.Ram Disk:>
```

Your display should be something similar if it's all worked. Notice how the output from the program runs into the prompt (the last line). We'll fix this soon.

1.6 Understanding a Simple Program

Understanding a Simple Program
****

To understand the example program we need to understand quite a few things. The observant amongst you will have noticed that all it does is print out a message, and that message was part of a line we wrote in the program. The first thing to do is see how to change this message.

1.7 Changing the Message

Changing the Message
==========

Edit the file so that line contains a different message between the two ' characters and compile it again using the same procedure as before. Don't use any ' characters except those around the message. If all went well, when you run the program again it should produce a different message. If something went wrong, compare the contents of your file with the original and make sure the only difference is the message between the ' characters.

1.8 Tinkering with the example

Tinkering with the example

Simple tinkering is a good way to learn for yourself so it is encouraged on these simple examples. Don't stray too far, though, and if you start getting confused return to the proper example pretty sharpish!

1.9 Brief overview

Brief overview

We'll look in detail at the important parts of the program in the following sections, but we need first to get a glimpse of the whole picture. Here's a brief description of some fundamental concepts:

* Procedures: We defined a procedure called main and used the (built-in) procedure WriteF. A procedure can be thought of as a small program with a name.

* Parameters: The message in parentheses after WriteF in our program is the parameter to WriteF. This is the data which the procedure should use.

* Strings: The message we passed to WriteF was a series of characters enclosed in ' characters. This is known as a string.

1.10 Procedures

```
Procedures
==========
```

As mentioned above, a procedure can be thought of as a small program with a name. In fact, when an E program is run the procedure called main is executed. Therefore, if your E program is going to do anything you must define a main procedure. Other (built-in or user-defined) procedures may be run (or called) from this procedure (as we did WriteF in the example). For instance, if the procedure fred calls the procedure barney the code (or mini-program) associated with barney is executed. This may involve calls to other procedures, and when the execution of this code is complete the next piece of code in the procedure fred is executed (and this is generally the next line of the procedure). When the end of the procedure main has been reached the program has finished. However, lots can happen between the beginning and end of a procedure, and sometimes the program may never get to finish. Alternatively, the program may crash, causing strange things to happen to your computer.

1.11 Procedure Definition

```
Procedure Definition
--------------------
```

Procedures are defined using the keyword PROC, followed by the new procedure's name (starting with a lowercase letter), a description of the parameters it takes (in parentheses), a series of lines forming the code of the procedure and then the keyword ENDPROC. Look at the example program again to identify the various parts. See The code.

1.12 Procedure Execution

```
Procedure Execution
-------------------
```

Procedures can be called (or executed) from within the code part of another procedure. You do this by giving its name, followed by some data

in parentheses. Look at the call to WriteF in the example program. See
The code.

1.13 Extending the example

```
Extending the example
---------------------
```

Here's how we could change the example program to define another procedure:

```
PROC main()
  WriteF('My first program')
  fred()
ENDPROC

PROC fred()
  WriteF('...slightly improved')
ENDPROC
```

This may seem complicated, but in fact it's very simple. All we've done is define a second procedure called fred which is just like the original program--it outputs a message. We've called this procedure in the main procedure just after the line which outputs the original message. Therefore, the message in fred is output after this message. Compile the program as before and run it so you don't have to take my word for it.

1.14 Parameters

```
Parameters
==========
```

Generally we want procedures to work with particular data. In our example we wanted the WriteF procedure to work on a particular message. We passed the message as a parameter (or argument) to WriteF by putting it between the parentheses (the (and) characters) that follow the procedure name. When we called the fred procedure, however, we did not require it to use any data so the parentheses were left empty.

When defining a procedure we define how much and what type of data we want it to work on, and when calling a procedure we give the specific data it should use. Notice that the procedure fred (like the procedure main) has empty parentheses in its definition. This means that the procedure cannot be given any data as parameters when it is called. Before we can define our own procedure that takes parameters we must learn about variables. We'll do this in the next chapter. See
Global and local variables.

1.15 Strings

```
Strings
=======
```

A series of characters between two ’ characters is known as a string. Almost any character can be used in a string, although the \ and ’ characters have a special meaning. For instance, a linefeed is denoted by the two characters \n. We now know how to stop the message running into the prompt. Change the program to be:

```
PROC main()
  WriteF(’My first program\n’)
  fred()
ENDPROC

PROC fred()
  WriteF(’...slightly improved\n’)
ENDPROC
```

Compile it as before, and run it. You should notice that the messages now appear on lines by themselves, and the second message is separated from the prompt which follows it. We have therefore cured the linefeed problem we spotted earlier (see Execution).

1.16 Style Reuse and Readability

```
Style, Reuse and Readability
============================
```

The example has grown into two procedures, one called main and one called fred. However, we could get by with only one procedure:

```
PROC main()
  WriteF(’My first program\n’)
  WriteF(’...slightly improved\n’)
ENDPROC
```

What we’ve done is replace the call to the procedure fred with the code it represents (this is called inlining the procedure). In fact, almost all programs can be easily re-written to eliminate all but the main procedure. However, splitting a program up using procedures normally results in more readable code. It is also helpful to name your procedures so that their function is apparent, so our procedure fred should probably have been named message or something similar. A well-written program in this style can read just like English (or any other spoken language).

Another reason for having procedures is to reuse code, rather than having to write it out every time you use it. Imagine you wanted to print the same, long message fairly often in your program--you’d either have to write it all out every time, or you could write it once in a procedure and call this procedure when you wanted the message printed. Using a procedure also has the benefit of having only one copy of the message to

change, should it ever need changing.

1.17 The Simple Program

The Simple Program
==========

The simple program should now (hopefully) seem simple. The only bit that hasn't been explained is the built-in procedure WriteF. E has many built-in procedures and later we'll meet some of them in detail. The first thing we need to do, though, is manipulate data. This is really what a computer does all the time--it accepts data from some source (possibly the user), manipulates it in some way (possibly storing it somewhere, too) and outputs new data (usually to a screen or printer). The simple example program did all this, except the first two stages were rather trivial. You told the computer to execute the compiled program (this was some user input) and the real data (the message to be printed) was retrieved from the program. This data was manipulated by passing it as a parameter to WriteF, which then did some clever stuff to print it on the screen. To do our own manipulation of data we need to learn about variables and expressions.

1.18 Variables and Expressions

Variables and Expressions
****

Anybody who's done any school algebra will probably know what a variable is--it's just a named piece of data. In algebra the data is usually a number, but in E it can be all sorts of things (e.g., a string). The manipulation of data like the addition of two numbers is known as an expression. The result of an expression can be used to build bigger expressions. For instance, 1+2 is an expression, and so is 6-(1+2). The good thing is you can use variables in place of data in expressions, so if x represents the number 1 and y represents 5, then the expression y-x represents the number 4. In the next two sections we'll look at what kind of variables you can define and what the different sorts of expressions are.

1.19 Variables

Variables
=========

Variables in E can hold many different kinds of data (called types). However, before a variable can be used it must be defined, and this is known as declaring the variable. A variable declaration also decides whether the variable is available for the whole program or just during the code of a procedure (i.e., whether the variable is global or local). Finally, the data stored in a variable can be changed using assignments. The following sections discuss these topics in slightly more detail.

1.20 Variable types

Variable types

In E a variable is a storage place for data (and this storage is part of the Amiga's RAM). Different kinds of data may require different amounts of storage. However, data can be grouped together in types, and two pieces of data from the same type require the same amount of storage. Every variable has an associated type and this dictates the maximum amount of storage it uses. Most commonly, variables in E store data from the type LONG. This type contains the integers from -2,147,483,648 to 2,147,483,647, so is normally more than sufficient. There are other types, such as INT and LIST, and more complex things to do with types, but for now knowing about LONG is enough.

1.21 Variable declaration

Variable declaration

Variables must be declared before they can be used. They are declared using the DEF keyword followed by a (comma-separated) list of the names of the variables to be declared. These variables will all have type LONG (later we will see how to declare variables with other types). Some examples will hopefully make things clearer:

```
DEF x

DEF a, b, c
```

The first line declares the single variable x, whilst the second declares the variables a, b and c all in one go.

1.22 Assignment

Assignment

The data stored by variables can be changed and this is normally done using assignments. An assignment is formed using the variable's name and an expression denoting the new data it is to store. The symbol := separates the variable from the expression. For example, the following code stores the number two in the variable x. The left-hand side of the := is the name of the variable to be affected (x in this case) and the right-hand side is an expression denoting the new value (simply the number two in this case).

```
x := 2
```

The following, more complex example uses the value stored in the variable before the assignment as part of the expression for the new data. The value of the expression on the right-hand side of the := is the value stored in the variable x plus one. This value is then stored in x, over-writing the previous data. (So, the overall effect is that x is incremented.)

```
x := x + 1
```

This may be clearer in the next example which does not change the data stored in x. In fact, this piece of code is just a waste of CPU time, since all it does is look up the value stored in x and store it back there!

```
x := x
```

1.23 Global and local variables

Global and local variables (and procedure parameters)

There are two kinds of variable: global and local. Data stored by global variables can be read and changed by all procedures, but data stored by local variables can only be accessed by the procedure to which they are local. Global variables must be declared before the first procedure definition. Local variables are declared within the procedure to which they are local (i.e., between the PROC and ENDPROC). For example, the following code declares a global variable w and local variables x and y.

```
DEF w
```

```
PROC main()
  DEF x
  x:=2
  w:=1
  fred()
ENDPROC

PROC fred()
  DEF y
  y:=3
  w:=2
ENDPROC
```

The variable x is local to the procedure main, and y is local to fred. The procedures main and fred can read and alter the value of the global variable w, but fred cannot read or alter the value of x (since that variable is local to main). Similarly, main cannot read or alter y.

The local variables of one procedure are, therefore, completely different to the local variables of another procedure. For this reason they can share the same names without confusion. So, in the above example, the local variable y in fred could have been called x and the program would have done exactly the same thing.

```
DEF w

PROC main()
  DEF x
  x:=2
  w:=1
  fred()
ENDPROC

PROC fred()
  DEF x
  x:=3
  w:=2
ENDPROC
```

This works because the x in the assignment in fred can refer only to the local variable x of fred (the x in main is local to main so cannot be accessed from fred).

If a local variable for a procedure has the same name as a global variable then in the rest of the procedure the name refers only to the local variable. Therefore, the global variable cannot be accessed in the procedure, and this is called descoping the global variable.

The parameters of a procedure are local variables for that procedure. We've seen how to pass values as parameters when a procedure is called (the use of WriteF in the example), but until now we haven't been able to define a procedure which takes parameters. Now we know a bit about variables we can have a go:

```
DEF y

PROC onemore(x)
```

```
    y:=x+1
  ENDPROC
```

This isn't a complete program so don't try to compile it. Basically, we've declared a variable y (which will be of type LONG) and a procedure onemore. The procedure is defined with a parameter x, and this is just like a (local) variable declaration. When onemore is called a parameter must be supplied, and this value is stored in the (local) variable x before execution of onemore's code. The code stores the value of x plus one in the (global) variable y. The following are some examples of calling onemore:

```
    onemore(120)
    onemore(52+34)
    onemore(y)
```

A procedure can be defined to take any number of parameters. Below, the procedure addthem is defined to take two parameters, a and b, so it must therefore be called with two parameters. Notice that values stored by the parameter variables (a and b) can be changed within the code of the procedure, since they are just like local variables for the procedure. (The only real difference between local and parameter variables is that parameter variables are initialised with the values supplied as parameters when the procedure is called.)

```
  DEF y

  PROC addthem(a, b)
    a:=a+2
    y:=a*b
  ENDPROC
```

The following are some examples of calling addthem:

```
    addthem(120,-20)
    addthem(52,34)
    addthem(y,y)
```

Global variables are, by default, initialised to zero. Parameter variables are, of course, initialised by the actual values passed as parameters when a procedure is called. However, local variables are not initialised. This means that a local variable will contain a fairly random value when the code of a procedure is first executed. It is the responsibility of the programmer to ensure no assumptions are made about the value of local variables before they have been initialised. The obvious way to initialise a local variable is using an assignment, but there is also a way of giving an initialisation value as part of the declaration (see Initialised Declarations). Initialisation of variables is often very important, and is a common reason why programs go wrong.

1.24 Changing the example

```
Changing the example
--------------------
```

Before we change the example we must learn something about WriteF. We already know that the characters \n in a string mean a linefeed. However, there are several other important combinations of characters in a string, and some are special to procedures like WriteF. One such combination is \d, which is easier to describe after we've seen the changed example.

```
PROC main()
  WriteF('My first program\n')
  fred()
ENDPROC

PROC fred()
  WriteF('...brought to you by the number \d\n', 236)
ENDPROC
```

You might be able to guess what happens, but compile it and try it out anyway. If everything's worked you should see that the second message prints out the number that was passed as the second parameter to WriteF. That's what the \d combination does--it marks the place in the string where the number should be printed. Here's the output the example should generate:

```
My first program
...brought to you by the number 236
```

Try this next change:

```
PROC main()
  WriteF('My first program\n')
  fred()
ENDPROC

PROC fred()
  WriteF('...the number \d is quite nice\n', 16)
ENDPROC
```

This is very similar, and just shows that the \d really does mark the place where the number is printed. Again, here's the output it should generate:

```
My first program
...the number 16 is quite nice
```

We'll now try printing two numbers.

```
PROC main()
  WriteF('My first program\n')
  fred()
ENDPROC

PROC fred()
  WriteF('...brought to you by the numbers \d and \d\n', 16, 236)
ENDPROC
```

Because we're printing two numbers we need two lots of \d, and we need to

supply two numbers as parameters in the order in which we want them to be printed. The number 16 will therefore be printed before the word 'and' and before the number 236. Here's the output:

```
My first program
...brought to you by the numbers 16 and 236
```

We can now make a big step forward and pass the numbers as parameters to the procedure fred. Just look at the differences between this next example and the previous one.

```
PROC main()
  WriteF('My first program\n')
  fred(16, 236)
ENDPROC

PROC fred(a,b)
  WriteF('...brought to you by the numbers \d and \d\n', a,b)
ENDPROC
```

This time we pass the (local) variables a and b to WriteF. This is exactly the same as passing the values they store (which is what the previous example did), and so the output will be the same. In the next section we'll manipulate the variables by doing some arithmetic with a and b, and get WriteF to print the results.

1.25 Expressions

```
Expressions
===========
```

The E language includes the normal mathematical and logical operators. These operators are combined with values (usually in variables) to give expressions which yield new values. The following sections discuss this topic in more detail.

1.26 Mathematics

```
Mathematics
-----------
```

All the standard mathematical operators are supported in E. You can do addition, subtraction, multiplication and division. Other functions such as sine, modulus and square-root can also be used as they are part of the Amiga system libraries, but we only need to know about simple mathematics

at the moment. The + character is used for addition, - for subtraction, * for multiplication (it's the closest you can get to a multiplication sign on a keyboard without using the letter x), and / for division (be careful not to confuse the \ used in strings with / used for division). The following are examples of expressions:

```
1+2+3+4
15-5
5*2
330/33
-10+20
3*3+1
```

Each of these expressions yields ten as its result. The last example is very carefully written to get the precedence correct (see Precedence and grouping).

All the above expressions use integer operators, so they manipulate integers, giving integers as results. Floating-point numbers are also supported by E, but using them is quite complicated (see Floating-Point Numbers). (Floating-point numbers can represent both very small fractions and very large integers, but they have a limited accuracy, i.e., a limited number of significant digits.)

1.27 Logic and comparison

```
Logic and comparison
--------------------
```

Logic lies at the very heart of a computer. They rarely guess what to do next; instead they rely on hard facts and precise reasoning. Consider the password protection on most games. The computer must decide whether you entered the correct number or word before it lets you play the game. When you play the game it's constantly making decisions: did your laser hit the alien?, have you got any lives left?, etc. Logic controls the operation of a program.

In E, the constants TRUE and FALSE represent the truth values true and false (respectively), and the operators AND and OR are the standard logic operators. The comparison operators are = (equal to), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to) and <> (not equal to). All the following expressions are true:

```
TRUE
TRUE AND TRUE
TRUE OR FALSE
1=1
2>1
3<>0
```

And these are all false:

```
FALSE
TRUE AND FALSE
```

```
FALSE OR FALSE
0=2
2<1
(2<1) AND (-1=0)
```

The last example must use parentheses. We'll see why in the next section (it's to do with precedence, again).

The truth values TRUE and FALSE are actually numbers. This is how the logic system works in E. TRUE is the number -1 and FALSE is zero. The logic operators AND and OR expect such numbers as their parameters. In fact, the AND and OR operators are really bit-wise operators (see Bitwise AND and OR), so most of the time any non-zero number is taken to be TRUE. It can sometimes be convenient to rely on this knowledge, although most of the time it is preferable (and more readable) to use a slightly more explicit form. Also, these facts can cause a few subtle problems as we shall see in the next section.

1.28 Precedence and grouping

Precedence and grouping

At school most of us are taught that multiplications must be done before additions in a sum. In E it's different--there is no operator precedence, and the normal order in which the operations are performed is left-to-right, just like the expression is written. This means that expressions like 1+3*3 do not give the results a mathematician might expect. In fact, 1+3*3 represents the number 12 in E. This is because the addition, 1+3, is done before the multiplication, since it occurs before the multiplication. If the multiplication were written before the addition it would be done first (like we would normally expect). Therefore, 3*3+1 represents the number 10 in E and in school mathematics.

To overcome this difference we can use parentheses to group the expression. If we'd written 1+(3*3) the result would be 10. This is because we've forced E to do the multiplication first. Although this may seem troublesome to begin with, it's actually a lot better than learning a lot of rules for deciding which operator is done first (in C this can be a real pain, and you usually end up writing the brackets in just to be sure!).

The logic examples above contained the expression:

```
(2<1) AND (-1=0)
```

This expression was false. If we'd left the parentheses out, it would have been:

```
2<1 AND -1=0
```

This is actually interpreted the same as:

```
((2<1) AND -1) = 0
```

Now the number -1 shouldn't really be used to represent a truth value with AND, but we do know that TRUE is the number -1, so E will make sense of this and the E compiler won't complain. We will soon see how AND and OR really work (see Bitwise AND and OR), but for now we'll just work out what E would calculate for this expression:

1. Two is not less than one so 2<1 can be replaced by FALSE.

   (FALSE AND -1) = 0

2. TRUE is -1 so we can replace -1 by TRUE.

   (FALSE AND TRUE) = 0

3. FALSE AND TRUE is FALSE.

   (FALSE) = 0

4. FALSE is really the number zero, so we can replace it with zero.

   0 = 0

5. Zero is equal to zero, so the expression is TRUE.

   TRUE

So E calculates the expression to be true. But the original expression (with parentheses) was false. Bracketing is therefore very important! It is also very easy to do correctly.

1.29 Program Flow Control

Program Flow Control
****

A computer program often needs to repeatedly execute a series of statements or execute different statements according to the result of some decision. For example, a program to print all the numbers between one and a thousand would be very long and tedious to write if each print statement had to be given individually--it would be much better to use a variable and repeatedly print its value and increment it.

Another aspect of flow control is choosing between different pieces of code to execute. For instance, if something goes wrong a program may need to decide whether to continue or print an error message and stop--this part of a program is a typical example of a conditional block.

Conditional Block
Loops

1.30 Conditional Block

```
Conditional Block
=================

   There are two kinds of conditional block: IF and SELECT.  Examples of
these blocks are given below as fragments of E code (i.e., the examples
are not complete E programs).

       IF x>0
         x:=x+1
         WriteF('Increment: x is now \d\n', x)
       ELSEIF x<0
         x:=x-1
         WriteF('Decrement: x is now \d\n', x)
       ELSE
         WriteF('Zero: x is 0\n')
       ENDIF

In the above IF block, the first part checks if the value of x is greater
than zero, and, if it is, x is incremented and the new value is printed
(with a message saying it was incremented).  The program will then skip
the rest of the block, and will execute the statements which follow the
ENDIF.  If, however, x it is not greater than zero the ELSEIF part is
checked, so if x is less than zero it will be decremented and printed, and
the rest of the block is skipped.  If x is not greater than zero and not
less than zero the statements in the ELSE part are executed, so a message
saying x is zero is printed.  The IF conditional is described in more
detail below.


  IF block
  IF expression

       SELECT x
         CASE 0
           WriteF('x is zero\n')
         CASE 10
           WriteF('x is ten\n')
         CASE -2
           WriteF('x is -2\n')
         DEFAULT
           WriteF('x is not zero, ten or -2\n')
       ENDSELECT

The SELECT block is similar to the IF block--it does different things
depending on the value of x.  However, x is only checked against specific
values, given in the series of CASE statements.  If it is not any of these
values the DEFAULT part is executed.

   There's also a variation on the SELECT block (known as the SELECT..OF
block) which matches ranges of values and is quite fast.  The two kinds of
SELECT block are described in more detail below.


  SELECT block
```

```
  SELECT..OF block
```

1.31 IF block

```
IF block
--------

   The IF block has the following form (the bits like expression are
descriptions of the kinds of E code which is allowed at that point--they
are not proper E code):

       IF expressionA
         statementsA
       ELSEIF expressionB
         statementsB
       ELSE
         statementsC
       ENDIF

This block means:

   * If expressionA is true (i.e., represents TRUE or any non-zero
     number) the code denoted by statementsA is executed.

   * If expressionA is false (i.e., represents FALSE or zero) and
     expressionB is true the statementsB part is executed.

   * If both expressionA and expressionB are false the statementsC
     part is executed.

There does not need to be an ELSE part but if one is present it must be
the last part (immediately before the ENDIF).  Also, there can be any
number of ELSEIF parts between the IF and ELSE parts.

   An alternative to this vertical form (where each part is on a separate
line) is the horizontal form:

       IF expression THEN statementA ELSE statementB

This has the disadvantage of no ELSEIF parts and having to cram everything
onto a single line.  Notice the presence of the THEN keyword to separate the
expression and statementA.  This horizontal form is closely related to
the IF expression, which is described below (see IF expression).

   To help make things clearer here are a number of E code fragments which
illustrate the allowable IF blocks:

       IF x>0 THEN x:=x+1 ELSE x:=0

       IF x>0
         x:=x+1
       ELSE
         x:=0
       ENDIF
```

```
IF x=0 THEN WriteF('x is zero\n')

IF x=0
  WriteF('x is zero\n')
ENDIF

IF x<0
  Write('Negative x\n')
ELSEIF x>2000
  Write('Too big x\n')
ELSEIF (x=2000) OR (x=0)
  Write('Worrying x\n')
ENDIF

IF x>0
  IF x>2000
    WriteF('Big x\n')
  ELSE
    WriteF('OK x\n')
  ENDIF
ELSE
  IF x<-800 THEN WriteF('Small x\n') ELSE Write('Negative OK x')
ENDIF
```

In the last example there are nested IF blocks (i.e., an IF block within an IF block). There is no ambiguity in which ELSE or ELSEIF parts belong to which IF block because the beginning and end of the IF blocks are clearly marked. For instance, the first ELSE line can be interpreted only as being part of the innermost IF block.

As a matter of style the conditions on the IF and ELSEIF parts should not overlap (i.e., at most one of the conditions should be true). If they do, however, the first one will take precedence. Therefore, the following two fragments of E code do the same thing:

```
IF x>0
  WriteF('x is bigger than zero\n')
ELSEIF x>200
  WriteF('x is bigger than 200\n')
ELSE
  WriteF('x is too small\n')
ENDIF

IF x>0
  WriteF('x is bigger than zero\n')
ELSE
  WriteF('x is too small\n')
ENDIF
```

The ELSEIF part of the first fragment checks whether x is greater than 200. But, if it is, the check in the IF part would have been true (x is certainly greater than zero if it's greater than 200), and so only the code in the IF part is executed. The whole IF block behaves as if the ELSEIF was not there.

1.32 IF expression

```
IF expression
-------------
```

IF is such a commonly used construction that there is also an IF expression. The IF block is a statement and it controls which lines of code are executed, whereas the IF expression is an expression and it controls its own value. For example, the following IF block:

```
IF x>0
  y:=x+1
ELSE
  y:=0
ENDIF
```

can be written more succinctly using an IF expression:

```
y:=(IF x>0 THEN x+1 ELSE 0)
```

The parentheses are unnecessary but they help to make the example more readable. Since the IF block is just choosing between two assignments to y it isn't really the lines of code that are different (they are both assignments), rather it is the values that are assigned to y that are different. The IF expression makes this similarity very clear. It chooses the value to be assigned in just the same way that the IF block choose the assignment.

The IF expression has the following form:

```
IF exp THEN expA ELSE expB
```

As you can see, IF expressions are written like the horizontal form of the IF block. However, there must be an ELSE part and there can be no ELSEIF parts. This means that the expression will always have a value (either expA or expB, depending on the value of exp), and it isn't cluttered with lots of cases.

Don't worry too much about IF expressions, since there are only useful in a handful of cases and can always be rewritten as a more wordy IF block. Having said that they are very elegant and a lot more readable than the equivalent IF block.

1.33 SELECT block

```
SELECT block
------------
```

The SELECT block has the following form:

```
SELECT variable
CASE expressionA
  statementsA
```

```
       CASE expressionB
         statementsB
       DEFAULT
         statementsC
       ENDSELECT
```

The value of the selection variable (denoted by variable in the SELECT part) is compared with the value of the expression in each of the CASE parts in turn. If there's a match, the statements in the (first) matching CASE part are executed. There can be any number of CASE parts between the SELECT and DEFAULT parts. If there is no match, the statements in the DEFAULT part are executed. There does not need to be a DEFAULT part but if one is present it must be the last part (immediately before the ENDSELECT).

It should be clear that SELECT blocks can be rewritten as IF blocks, with the checks on the IF and ELSEIF parts being equality checks on the selection variable. For example, the following code fragments are equivalent:

```
       SELECT x
       CASE 22
         WriteF('x is 22\n')
       CASE (y+z)/2
         WriteF('x is (y+x)/2\n')
       DEFAULT
         WriteF('x isn't anything significant\n')
       ENDSELECT

       IF x=22
         WriteF('x is 22\n')
       ELSEIF x=(y+z)/2
         WriteF('x is (y+x)/2\n')
       ELSE
         WriteF('x isn't anything significant\n')
       ENDIF
```

Notice that the IF and ELSEIF parts come from the CASE parts, the ELSE part comes from the DEFAULT part, and the order of the parts is preserved. The advantage of the SELECT block is that it's much easier to see that the value of x is being tested all the time, and also we don't have to keep writing x= in the checks.

1.34 SELECT..OF block

```
SELECT..OF block
----------------
```

The SELECT..OF block is a bit more complicated than the normal SELECT block, but can be very useful. It has the following form:

```
       SELECT maxrange OF expression
       CASE constA
         statementsA
```

```
CASE constB1 TO constB2
  statementsB
CASE range1, range2
  statementsC
DEFAULT
  statementsD
ENDSELECT
```

The value to be matched is expression, which can be any expression, not just a variable like in the normal SELECT block. However, the maxrange, constA, constB1 and constB2 must all be explicit numbers, i.e., constants (see Constants). maxrange must be a positive constant and the other constants must all be between zero and maxrange (including zero but excluding maxrange).

The CASE values to be matched are specified using ranges. A simple range is a single constant (the first CASE above). The more general range is shown in the second CASE, using the TO keyword (constB2 must be greater than constB1). A general CASE in the SELECT..OF block can specify a number of possible ranges to match against by separating each range with a comma, as in the third CASE above. For example, the following CASE lines are equivalent and can be used to match any number from one to five (inclusive):

```
CASE 1 TO 5

CASE 1, 2, 3, 4, 5

CASE 1 TO 3, 3 TO 5

CASE 1, 2 TO 3, 4, 5

CASE 1, 5, 2, 4, 3

CASE 2 TO 3, 5, 1, 4
```

If the value of the expression is less than zero, greater than or equal to maxrange, or it does not match any of the constants in the CASE ranges, then the statements in the DEFAULT part are executed. Otherwise the statements in the first matching CASE part are executed. As in the normal SELECT block, there does not need to be a DEFAULT part.

The following SELECT..OF block prints the (numeric) day of the month nicely:

```
SELECT 32 OF day
CASE 1, 21, 31
  WriteF('The \dst day of the month\n', day)
CASE 2, 22
  WriteF('The \dnd day of the month\n', day)
CASE 3, 23
  WriteF('The \drd day of the month\n', day)
CASE 4 TO 20, 24 TO 30
  WriteF('The \dth day of the month\n', day)
DEFAULT
  WriteF('Error: invalid day=\d\n', day)
ENDSELECT
```

The maxrange for this block is 32, since 31 is the maximum of the values used in the CASE parts. If the value of day was 100, for instance, then the statements in the DEFAULT part would be executed, signalling an invalid day.

This example can be rewritten as an IF block:

```
IF (day=1) OR (day=21) OR (day=31)
  WriteF('The \dst day of the month\n', day)
ELSEIF (day=2) OR (day=22)
  WriteF('The \dnd day of the month\n', day)
ELSEIF (day=3) OR (day=23)
  WriteF('The \drd day of the month\n', day)
ELSEIF ((4<=day) AND (day<=20))  OR  ((24<=day) AND (day<=30))
  WriteF('The \dth day of the month\n', day)
ELSE
  WriteF('Error: invalid day=\d\n', day)
ENDIF
```

The comma separating two ranges in the CASE part has been replaced by an OR of two comparison expressions, and the TO range has been replaced by an AND of two comparisons. (It is worth noticing the careful bracketing of the resulting expressions.)

Clearly, the SELECT..OF block is much more readable than the equivalent IF block. It is also a lot faster, mainly because none of the comparisons present in IF block have to be done in the SELECT..OF version. Instead the value to be matched is used to immediately locate the correct CASE part. However, it's not all good news: the maxrange value directly affects the size of compiled executable, so it is recommended that SELECT..OF blocks be used only with small maxrange values. See the 'Reference Manual' for more details.

1.35 Loops

Loops
=====

Loops are all about making a program execute a series of statements over and over again. Probably the simplest loop to understand is the FOR loop. There are other kinds of loops, but they are easier to understand once we know how to use a FOR loop.

```
FOR loop
WHILE loop
REPEAT..UNTIL loop
```

1.36 FOR loop

FOR loop

If you want to write a program to print the numbers one to 100 you can either type each number and wear out your fingers, or you can use a single variable and a small FOR loop. Try compiling this E program (the space after the \d in the string is needed to separate the printed numbers):

```
PROC main()
  DEF x
  FOR x:=1 TO 100
    WriteF('\d ', x)
  ENDFOR
  WriteF('\n')
ENDPROC
```

When you run this you'll get all the numbers from one to 100 printed, just like we wanted. It works by using the (local) variable x to hold the number to be printed. The FOR loop starts off by setting the value of x to one (the bit that looks like an assignment). Then the statements between the FOR and ENDFOR lines are executed (so the value of x gets printed). When the program reaches the ENDFOR it increments x and checks to see if it is bigger than 100 (the limit we set with the TO part). If it is, the loop is finished and the statements after the ENDFOR are executed. If, however, it wasn't bigger than 100, the statements between the FOR and ENDFOR lines are executed all over again, and this time x is one bigger since it has been incremented. In fact, this program does exactly the same as the following program (the ... is not E code--it stands for the 97 other WriteF statements):

```
PROC main()
  WriteF('\d ', 1)
  WriteF('\d ', 2)
  ...
  WriteF('\d ', 100)
  WriteF('\n')
ENDPROC
```

The general form of the FOR loop is as follows:

```
  FOR var := expressionA TO expressionB STEP number
    statements
  ENDFOR
```

The var bit stands for the loop variable (in the example above this was x). The expressionA bit gives the start value for the loop variable and the expressionB bit gives the last allowable value for it. The STEP part allows you to specify the value (given by number) which is added to the loop variable on each loop. Unlike the values given for the start and end (which can be arbitrary expressions), the STEP value must be a constant (see Constants). The STEP value defaults to one if the STEP part is omitted (as in our example). Negative STEP values are allowed, but in this case the check used at the end of each loop is whether the loop variable is less than the value in the TO part. Zero is not allowed as the STEP value.

As with the IF block there is a horizontal form of a FOR loop:

```
FOR var := expA TO expB STEP expC DO statement
```

1.37 WHILE loop

```
WHILE loop
----------
```

The FOR loop used a loop variable and checked whether that variable had gone past its limit. A WHILE loop allows you to specify your own loop check. For instance, this program does the same as the program in the previous section:

```
PROC main()
  DEF x
  x:=1
  WHILE x<=100
    WriteF('\d ', x)
    x:=x+1
  ENDWHILE
  WriteF('\n')
ENDPROC
```

We've replaced the FOR loop with an initialisation of x and a WHILE loop with an extra statement to increment x. We can now see the inner workings of the FOR loop and, in fact, this is exactly how the FOR loop works.

It is important to know that our check, x<=100, is done before the loop statements are executed. This means that the loop statements might not even be executed once. For instance, if we'd made the check x>=100 it would be false at the beginning of the loop (since x is initialised to one in the assignment before the loop). Therefore, the loop would have terminated immediately and execution would pass straight to the statements after the ENDWHILE.

Here's a more complicated example:

```
PROC main()
  DEF x,y
  x:=1
  y:=2
  WHILE (x<10) AND (y<10)
    WriteF('x is \d and y is \d\n', x, y)
    x:=x+2
    y:=y+2
  ENDWHILE
ENDPROC
```

We've used two (local) variables this time. As soon as one of them is ten or more the loop is terminated. A bit of inspection of the code reveals that x is initialised to one, and keeps having two added to it. It will, therefore, always be an odd number. Similarly, y will always be even. The WHILE check shows that it won't print any numbers which are greater

than or equal to ten. From this and the fact that x starts at one and y at two we can decide that the last pair of numbers will be seven and eight. Run the program to confirm this. It should produce the following output:

```
x is 1 and y is 2
x is 3 and y is 4
x is 5 and y is 6
x is 7 and y is 8
```

Like the FOR loop, there is a horizontal form of the WHILE loop:

```
WHILE expression DO statement
```

Loop termination is always a big problem. FOR loops are guaranteed to eventually reach their limit (if you don't mess with the loop variable, that is). However, WHILE loops (and all other loops) may go on forever and never terminate. For example, if the loop check were 1<2 it would always be true and nothing the loop could do would prevent it being true! You must therefore take care that your loops terminate in some way if you want to program to finish. There is a sneaky way of terminating loops using the JUMP statement, but we'll ignore that for now.

1.38 REPEAT..UNTIL loop

REPEAT..UNTIL loop

A REPEAT..UNTIL loop is very similar to a WHILE loop. The only difference is where you specify the loop check, and when and how the check is performed. To illustrate this, here's the program from the previous two sections rewritten using a REPEAT..UNTIL loop (try to spot the subtle differences):

```
PROC main()
  DEF x
  x:=1
  REPEAT
    WriteF('\d ', x)
    x:=x+1
  UNTIL x>100
  WriteF('\n')
ENDPROC
```

Just as in the WHILE loop version we've got an initialisation of x and an extra statement in the loop to increment x. However, this time the loop check is specified at the end of the loop (in the UNTIL part), and the check is only performed at the end of each loop. This difference means that the code in a REPEAT..UNTIL loop will be executed at least once, whereas the code in a WHILE loop may never be executed. Also, the logical sense of the check follows the English: a REPEAT..UNTIL loop executes until the check is true, whereas the WHILE loop executes while the check is true. Therefore, the REPEAT..UNTIL loop executes while the check is false! This may seem confusing at first, but just remember to read the code as if it were English and you'll get the correct interpretation.

1.39 Summary

```
Summary
*******
```

This is the end of Part One, which was hopefully enough to get you started. If you've grasped the main concepts you are good position to attack Part Two, which covers the E language in more detail.

This is probably a good time to look at the different parts of one of the examples from the previous sections, since we've now used quite a bit of E. The following examination uses the WHILE loop example. Just to make things easier to follow, each line has been numbered (don't try to compile it with the line numbers on!).

```
 1.  PROC main()
 2.    DEF x,y
 3.    x:=1
 4.    y:=2
 5.    WHILE (x<10) AND (y<10)
 6.      WriteF('x is \d and y is \d\n', x, y)
 7.      x:=x+2
 8.      y:=y+2
 9.    ENDWHILE
10.  ENDPROC
```

Hopefully, you should be able to recognise all the features listed in the table below. If you don't then you might need to go back over the previous chapters, or find a much better programming guide than this!

Line(s)	Observation
1-10	The procedure definition.
1	The declaration of the procedure main, with no parameters.
2	The declaration of local variables x and y.
3, 4	Initialisation of x and y using assignment statements.
5-9	The WHILE loop.
5	The loop check for the WHILE loop using the logical operator AND, the comparison operator <, and parentheses to group the expression.
6	The call to the (built-in) procedure WriteF using parameters. Notice the string, the place holders for numbers, \d, and the linefeed, \n.

```
7, 8    Assignments to x and y, adding two to
        their values.

   9    The marker for the end of the WHILE loop.

  10    The marker for the end of the procedure.
```