The INTER-BASE Programming Guide

Programming with INTER-BASE on the BBC Micro

By

Martin T. Pickering

The INTER-BASE Programming Guide

Programming with INTER-BASE on the BBC Micro

First Edition March 1991
ISBN 0-946273-05-7

The INTER-BASE Programming Guide

Programming with INTER-BASE on the BBC Micro

By

Martin T. Pickering

Arlon House Publishing
Station Road • Kings Langley • Hertfordshire • WD4 8LF
Tel: (0923) 268328

Disclaimer

Due to the nature of computer programming, no responsibility can be accepted by the author, the publisher or the manufacturer of the software for any loss or damage caused either directly or indirectly by the use of information given in this book. Efforts have been made to ensure that the information is correct at the time of publication, but it is the responsibility of the user to ensure that data is properly stored and backups made in order to prevent loss of data due to programming errors or hardware failures.

Technical Assistance

This book is provided as an aid to learning programming in INTER-BASE and should greatly reduce the need for you to write for technical assistance. The publishers are unable to provide technical assistance relating to the contents of this book, but letters will be passed to the Author.

Trademarks

"BBC B" and "Master" refer to computers made by Acorn Computers Ltd for the British Broadcasting Company.

Thanks

Thanks are due to all employees of Computer Concepts but especially to Neville for putting up with my incessant telephone calls. Thanks also to my brother, Rob, who provided the help and hardware needed to create this book.

Published and printed in England by

Arlon House Publishing
Station Road
Kings Langley
Hertfordshire
WD4 8LF
Telephone (0923) 268328

Contents

About INTER-BASE

Although it has little real relevance to programming, it is interesting to know how INTER-BASE came to be the way it is.

With WordWise Plus and INTER-WORD selling literally tens of thousands it was the users that continually pushed for a compatible database program. Within Computer Concepts the need was also clearly seen for a good database which would meet a wide variety of needs and integrate with existing products. Database programs already on the market were mainly written in BASIC and, while they performed simple card index activities adequately, they did little else. A program from Computer Concepts would have to do much more, or there would be little point in creating it.

The success and flexibility of WordWise Plus indicated the usefulness of an integral programming language. This, together with lessons learned from traditional database programs such as DBase pointed the way forward for INTER-BASE.

As part of the ROM-LINK series of packages, INTER-BASE could provide the programmability to combine the actions of the INTER-WORD word processor, INTER-SHEET spreadsheet, and INTER-CHART graph plotting program. It was envisaged that activities such as combined word processing, invoicing and stock control would easily be achieved.

In retrospect it's easy to see that the aims were just too high. The BBC Micro, powerful as it was for the early 1980s had too many limitations for the ambitions held for INTER-BASE and the rest of the ROM-LINK family approaching the 1990s. With a limit of about 25K of memory and the majority of users running with just floppy disc drives, virtually every useful application ran into difficulty.

INTER-BASE itself started out being written as a 16K program – the maximum size which could fit into a programmable chip in a BBC Micro. Like INTER-WORD, it soon grew too large. INTER-WORD was introduced on a specially designed chip carrier to allow a 32K program to fit on a standard machine. INTER-BASE grew and grew until eventually an even more expensive 64K version had to be produced. As such INTER-BASE is and probably always will be the largest machine code program to be produced for the BBC Micro. It is hardly surprising then that it was released some two years later than the original planned date.

Not all of the 64K EPROM chip is occupied with the INTER-BASE programming language. When you start INTER-BASE, a pre-programmed card index system is entered and its menu appears on the screen. This allows users at least some operationality without the need for programming. This card index was written using the INTER-BASE programming language, but cleverly stored inside the same chip. This way you don't have to load the program from disc each time you switch on, and it leaves more of the computer's (limited) main memory free for data.

INTER-BASE meets virtually all the criteria which the programming team intended for it. It offers integration with the INTER-WORD, SHEET and CHART; it is an extremely comprehensive programming language with very powerful database commands and few limitations on record size or maximum number of records. But it has one major failing – users without programming ability can achieve little more with INTER-BASE than was possible with the original 'cheap and cheerful' programs available several years earlier at half the price. Specifically, they only have at their fingertips the pre-programmed card index system which, because of space constraints within the chip, uses only some of the facilities supported in the programming language.

For those willing to put effort into programming, especially those who actually enjoy programming almost for the sake of it, INTER-BASE is a marvellous package. Even when you have programmed INTER-BASE for many months you will still continue to reveal hidden depths of its design.

The BASIC connection

INTER-BASE is a complete programming language in its own right. However, inventing a whole new language has the distinct disadvantage that you have to teach everyone how to use it from scratch. The one language that every programmer with a BBC Micro knows is BASIC, or BBC BASIC to be more specific. For this reason INTER-BASE was deliberately designed to be as similar to BASIC as possible, without compromising too far.

If you don't already know at least a little about programming in BASIC or a similar language, you're probably going to find it hard work to understand this book. I strongly suggest that you take the time to work through one of the many books available which teach BBC BASIC programming - virtually everything you learn will be useful in programming INTER-BASE.

If you already know how to program in BASIC, you'll love INTER-BASE because it removes just about all the annoying aspects of the language and adds a whole host of new facilities. The next chapter highlights some of the differences between BBC BASIC and IBPL[1].

On release...

INTER-BASE was finally released in August 1987 as version 1.0A. In the final desperate rush to get the package onto the market (more to appease impatient customers than for sound commercial reasons), the accompanying manual was short – far too short. The original intention was to produce a four hundred page manual covering database philosophies, a programming tutorial and a complete command reference. However, after almost two years of well-intentioned delays while more and more features were added, it was a 'now or never' decision which forced it to the marketplace with a manual of about one hundred pages.

Once again, the advantage of hindsight allows us to see this as a mistake. If only an extra couple of months had been spent working on documentation, INTER-BASE would have been better received and would ultimately have sold in greater numbers. However, I think I would be correct in saying that to Computer Concepts the thought of a further delay at that stage of the project would have resulted in immediate termination; INTER-BASE would never have seen the light of day.

[1] IBPL stands for INTER-BASE Programming Language.

At this stage, INTER-BASE contained quite a few unintentional features![1] Almost a year went by before these "features" were eliminated in version 1.10A. Unfortunately (or otherwise) the author was lent this pre-release version and discovered a couple of new "features". Finally, in August 1988, version 2.0A was released together with a new Reference manual with more than 250 pages of information. Of these, over 200 refer to programming. Still less than the originally intended volume of documentation, but adequate for most users. It was because there was so much more to be said that this book came about.

It is important that you have the latest significant release of the software; there's no point wasting time on features which have already been ironed out. Computer Concepts are very good about upgrades. If you have a version of INTER-BASE earlier than 2.0A then give them a call. They will tell you how to upgrade to the latest version, including the new manual, for only a small charge.

Although the 2.0A chip contains a number of revisions over 1.0A, most of these are associated with the card index program rather than IBPL. To find out which version is fitted in your computer type:-
*HELP<RETURN>.

About this book

This book is intended to help you to write programs in IBPL and will supplement and clarify the information in the Reference Manual. It could never entirely replace it, nor could it avoid duplicating a lot of information already there. Some sections of the reference manual are very comprehensive, and some are not. I hope to expand upon those parts which the Reference Manual mentions "in passing" but to limit the information about those parts already treated in detail.

A large part of this book is given over to the reference section which contains the programming language keywords. The reasons for the apparent duplication of this "list" already in the Reference Manual are to provide an alphabetically ordered reference, to correct errors in the

[1] A 'feature' is the polite way to describe a bug - a programming error. Tell a programmer there are bugs in his software and he is immediately on the defensive; suggest that there may be some 'unintentional features', keeping your tongue firmly in your cheek, and the conversation has a good chance of continuing!

original syntax, to add more explanations and to provide at least one example for most keywords.

The Examples

This book concentrates heavily on examples. All of the examples are working programs. All of them were written and tested in INTER-BASE 0 then transferred directly to INTER-WORD text by means of the EXPORT command. I believe deeply in the use of examples instead of pages of explanation. You should be able to find an example for most applications you will need. If you can't find the example under the most likely keyword then it is probably under an associated keyword. The following chapters give fairly lengthy program examples which put into practice the information given in the reference section. It is recommended that you read through the reference section to get a "feel" for the commands available; then read the programming chapters and make an effort to understand the examples. It should be stressed, however, that the fastest way to learn is by doing!

Note: Where an example would normally print a result on screen, a typical printout is shown after the program.

The Example disc

Some of the examples shown in this book are included on an example disc available separately. Please contact the publishers for details of price and availability.

The example disc is available only as an 80 track 5.25", formatted for ADFS on the first side and DFS on the second. This combination provides the widest compatibility. We regret that other formats can not be supplied.

What you need to know

As already stated, it is assumed that you already have some familiarity with the BBC BASIC programming language, upon which IBPL is heavily based. If you do not have this knowledge, you will probably find it hard work to follow this book. One of the BBC BASIC tutorial books is highly recommended as the entry point for newcomers to programming. Public libraries – even small local ones – usually have a selection of these.

Many aspects of this book build upon the information provided in the INTER-BASE Reference Manual, 2nd Edition. It is important that you read the Reference Manual or at least have it close to hand. Particularly you should be familiar with the terminology of database structures and operations. You should know about fields, records, files, indexes, data types, sorting, searching, and so on. You should also know from the Reference Manual how to change between the Card Index menu and the IBPL menu, and how to enter and execute commands and programs.

BASIC differences

Some of the main differences between IBPL and BASIC are listed below:

1. IBPL uses no line numbers but relies upon simple destination "labels" (words preceded by a full stop). For those of you who have used BBC BASIC assembly language the method of labelling will be familiar.

2. IBPL has many more 'structured' programming facilities, including multi-line IF...THEN...ELSE, WHILE...ENDWHILE, CASE...ENDCASE.

3. IBPL has very comprehensive record facilities which can work in a similar way to BASIC's arrays (with different syntax), but are far more flexible. They allow any array to contain different numbers of elements at each level, and any item can be of any data type (string, real, int, or even another array).

4. IBPL strings may be any length and may include carriage returns (and all other ASCII codes). BASIC strings are limited to 255 characters which can be very limiting for many types of data.

5. IBPL programs are stored in memory in ordinary strings, usually in plain ASCII form. This allows more than one program or sub-program to reside in memory at the same time, and it allows a program to create or modify another program. One common use of this facility is for a main program to stay in memory and to load any one of a number of sub-programs into another string for use only when they are required, giving an almost unlimited program size.

6. IBPL offers a huge variety of string search and manipulation keywords. These are designed specifically for the type of searching and sorting which is required in database applications. operations which take complex programming in BASIC are incredibly simple in IBPL.

8. Whilst BASIC contains rudimentary file handling commands, IBPL contains sophisticated commands which make the construction of a database and sorted index a relatively simple task.

10. IBPL supports the use of calendar dates, allowing them to be entered, stored, added together, subtracted from one another and even printed in any one of a number of common formats.

12. IBPL uses DATA in the form of a string, not as a DATA statement.

13. IBPL allows use of sideways RAM (as fitted as standard in the BBC Master and Compact) to be used as a fast temporary disc storage.

14. IBPL allows programs to be stored in sideways RAM and even permanently in EPROMS.

15. INTER-BASE contains a very flexible full or partial-screen text editor which is not only available for entering and editing programs but is also available as a command within IBPL. Programs can invite the user to enter and edit long pieces of text while still remaining in control.

16. IBPL has many linking features allowing data to be transferred to or from other ROM-LINK programs. For example, an IBPL program can retrieve documents from within INTER-WORD and manipulate them.

If you are not very familiar with BASIC, the next chapter will lead you gently into the art of IBPL programming.

Simple program writing in IBPL

Switch on the computer and type *IB.PMENU <RETURN> (in upper case) to enter INTER-BASE 0.
Press <ESCAPE> to enter the editor which operates like a simple word processor. Text written in this editor can be MARKED, COPIED and DELETED just as in INTER-WORD or WORDWISE.

Before using the editor you might like to alter the screen display mode which is usually mode 7 by default. If you press <ESCAPE> to return to the menu and type 73<RETURN> (BBC B) or 7131<RETURN> (Master) then the editor will display your text in 80 column mode instead of 40 when you press <ESCAPE> again. If the screen flickers excessively, return to the menu and type *TV0,1<RETURN> then press <ESCAPE> twice.

A program may be written as plain text in INTER-BASE 0 without the need for line numbers (in fact line numbers may NOT be used).

The name of the program must be typed after selecting option <5>.
The default program name is P$. To run a program type P <RETURN>
or type RUN P$ <RETURN>. If you had chosen the program name to be, say, "JUNKPROG" then to run it type JUNKPROG <RETURN> or type
RUN JUNKPROG$ <RETURN>. At present, of course, there is no program.

The main body of a program must always begin with .START
The main body of a program may end with RETURN or ENDPROC or END (or with nothing provided that no sub-procedures follow it).
For reasons too complex to explain in this chapter it is safest to end with RETURN

Here is the simplest form of program for you to try:

Example

```
.START
CLS
INPUT"Please enter a number from 1 to 3000 "mynumber
PRINT"Your number is "mynumber
RETURN
```

Explanation

The first program line after .START clears the screen.

The next line prints the first message on the screen and waits for you to type a number. It will continue to wait until you press <RETURN>.

The second message is then printed on the screen, followed by your number.

The word "mynumber" is called a variable name. The variable "mynumber" is called a "real" variable and its contents may include a decimal point if you wish and may also have a negative sign.

When you type in your chosen number the computer assigns it to the variable "mynumber" and stores it in a specific place in computer memory.

Type the program into P$. From the menu type P and press <RETURN>.

Calculations

Suppose we needed to calculate miles per gallon from a year's diary of litres and miles. If the first journey of 244 miles used 28.6 litres then, with a calculator, we must enter (say) the following sequence:

2 4 4 / 2 8 . 6 x 4 . 5 5 =

To repeat this type of calculation many times will prove time-consuming.

On the computer keyboard we can achieve the same object by typing:

PRINT 244/28.6*4.55

Again, this method is tedious but if we use the programming capabilities of the computer the task can be simplified.

The following program will simplify this task slightly:

Re-typing the following examples is rather time-consuming and introduces the possibility of errors. As an alternative they are provided on an example disc. If you have the disc, please load the file "EXAMPLE" into INTER-BASE 0 P$.

All of the following programs are contained within the same file so, as you proceed with these numbered examples, to run each program you must delete those which precede it in P$. Only the first program will run. The rest will be ignored after the RETURN instruction. Should you make a mistake simply reload the complete EXAMPLE file back into P$.

If you don't have the example disc simply type each example in turn.

Example1

```
.START
INPUT"Miles ? "miles
INPUT"Litres ? "litres
PRINT"MPG="miles/litres*4.55
RETURN
```

Type P<RETURN> to run this first example.
This program must be run for every entry and is still tedious to use but the problem is solved quite easily by making it repeat by itself:

Press <ESCAPE> to edit P$. Delete example1. Press <ESCAPE>.
Type P<RETURN> to run example2.

Example2

```
.START
INPUT"Miles ? "miles
INPUT"Litres ? "litres
PRINT"MPG="miles/litres*4.55
GOTO "START"
```

Now the program will request miles, litres, print MPG then repeat the same thing again.

The use of "GOTO" in this manner is a very crude method of making a program "loop" and to stop the sequence you must press <ESCAPE>.

Program loops

Example3

```
.START
CLS:PRINT'''
REPEAT
   INPUT"Miles ? "miles
   INPUT"Litres ? "litres
   mpg=miles/litres*4.55
   PRINTmpg
UNTIL mpg>35
RETURN
```

The program loop created by REPEAT ... UNTIL will cause the relevant portion of the program to repeat until an answer greater than 35 MPG is achieved or until <ESCAPE> is pressed. This type of loop is called a "conditional" loop since it stops after a specified condition is fulfilled.

Another method, if you know how may entries you have, is to use a FOR ... NEXT loop:

Example4

```
.START
CLS:PRINT'''
INPUT"How many entries ";number%
FOR X%=1 TO number%
   INPUT"Miles ? "miles
   INPUT"Litres ? "litres
PRINT"MPG="miles/litres*4.55
NEXT
RETURN
```

This example requests the number of entries and allocates this quantity to the variable "number%". Since the number must be a whole number we can use an "integer variable" which is designated by the percent symbol "%". In a large program the use of an "integer variable" instead of a "real variable" saves valuable memory space and can increase the running speed of the program.

Another integer variable, X%, is used to keep track of how many times the program loop is repeated.

We can make use of the additional information to help us calculate the overall average MPG. However, it is necessary to introduce a variable to hold the sub total mpg figure. For this any name would do but I chose "sum". Note that "sum" must be initialised by making it zero.

Example5

```
.START
CLS:PRINT'''
sum=0
INPUT"How many entries ? "number%
FOR X%=1 TO number%
   INPUT"Miles ? "miles
   INPUT"Litres ? "litres
   gals=litres/4.55
   mpg=miles/gals
   PRINT"MPG="mpg
   sum=sum+mpg
NEXT
PRINT"Average MPG= "sum/number%
RETURN
```

It is also possible to use REPEAT ... UNTIL to refine the program further so the number of entries is not required. We must initialise count% to zero. (This type of initialisation must be performed *before* a variable is used on the right hand side of an = sign). We also need to keep a count of the number of entries.

Example6

```
.START
ON ERROR:ON ERROR OFF:PRINT"OOPS":RETURN
CLS:PRINT'''
sum=0
count%=0
REPEAT
   INPUT"Miles ? "miles
   IF miles=0 THEN GOTO hop
   INPUT"Litres ? "litres
   gals=litres/4.55
   mpg=miles/gals
   PRINT"MPG="mpg
   sum=sum+mpg
   count%=count%+1
```

```
.hop
UNTIL miles=0
PRINT"Average MPG= "sum/count%
ON ERROR OFF
RETURN
```

The IF ... THEN condition causes the program to jump to the label ".hop" if miles=0 (or if <RETURN> alone is pressed). Since the UNTIL miles=0 condition is fulfilled the program stops. If <RETURN> is pressed the first time the loop is executed then a "division by zero" error will be announced because count% is still zero.

To combat errors such as this, the ON ERROR statement is used. Note that, when an error is encountered, it is important to turn ON ERROR OFF. Equally important is to do this, also, at the end of the program.

Part of the skill in programming lies in the presentation so, just for fun, let's introduce some graphics into this program!

The following example selects the high quality graphics mode 0 and draws a circle. A text WINDOW is designated inside the circle to constrain the print statements to that small area. At the end of the program the window is cancelled by VDU26.

Notice how the <TAB> key is used to indent part of the program. You do not have to do this but it helps to highlight the start and end of loops and is particularly useful when one loop occurs within another (called "nested loops").

Example7

```
.START
REM Example CIRCLE
MODE 0
S=SIN(RAD12):C=COS(RAD12):A=0:B=R
X=600:Y=500:R=300
MOVE X,Y+R
FOR A%=1 TO 31
   Z=A*C+B*S
   B=B*C-A*S
   A=Z
   DRAW X+Z,Y+B
NEXT
```

```
WINDOW 28,12,24,4

ON ERROR:ON ERROR OFF:PRINT"OOPS":RETURN
sum=0
count%=0
REPEAT
   INPUT"Miles ? "miles
   IF miles=0 THEN GOTO hop
   INPUT"Litres ? "litres
   gals=litres/4.55
   mpg=miles/gals
   PRINT"MPG="mpg
   sum=sum+mpg
   count%=count%+1
.hop
UNTIL miles=0
PRINT"Average MPG= "sum/count%
VDU26
ON ERROR OFF
RETURN
```

The circle routine can be used by itself for experimentation. There is no harm in changing some of the numbers to see what happens!

Try substituting PLOT 85,X+Z,Y+B instead of DRAW X+Z,Y+B.

Strings

So far we have dealt with numbers but IBPL has very powerful commands for dealing with text. We call a line of characters with no particular numeric value a "string". Of course it is not feasible to multiply or divide strings but you can certainly add and subtract them.

Example8

```
.START
INPUT"Please enter your last name "lastname$
INPUT"Please enter your first name "firstname$
INPUT"Please enter your full address "addr$
H$=firstname$+" "+lastname$+",|M"+addr$
PRINT H$
RETURN
```

This program adds together the three strings, putting a space between first and last names and putting a carriage return after them. One serious drawback, which becomes obvious when you try the program, is that it accepts only the first line of your address. This is because the INPUT function stops as soon as you press <RETURN>.

To combat this effect we can introduce a loop which will terminate only after <RETURN> has been pressed twice in succession.

Example9

```
.START
INPUT"Please enter your last name "lastname$
INPUT"Please enter your first name "firstname$
addr$=""
PRINT"Please enter your full address "a$
REPEAT
   INPUTLINE a$
   addr$=+a$+"|M"
UNTIL a$=""
H$=firstname$+" "+lastname$+",|M"+addr$
PRINT H$
RETURN
```

Since the INPUT function does not accept punctuation such as commas we have used INPUTLINE instead. By using INPUTLINE within a loop it is possible to put each address line into the string "a$" and add it to the string "addr$" together with a carriage return "|M".

Although the function INPUTLINE sees <RETURN> as the end of a line it does not include a carriage return in the string itself, so we must explicitly add one.

There are many string handling commands available in IBPL but, since there are very comprehensive examples in the Reference Section, no more specific examples will be given, here.
The previous examples dealt with printing to the screen only. For most applications it is useful to have a copy on paper. If you have a printer connected to the PRINTER port on the computer you can use the commands VDU2 to send the output to the printer and VDU3 to stop it.

Example10

```
.START
ON ERROR:ON ERROR OFF:VDU3:PRINT"OOPS":RETURN
CLS:PRINT'''
sum=0
count%=0
PRINT"Ensure printer is switched on!"
PRINT"Then press a key."
G=GET
VDU2:PRINT"miles litres MPG":VDU3
REPEAT
   INPUT"Miles ? "miles
   IF miles=0 THEN GOTO hop
   INPUT"Litres ? "litres
   gals=litres/4.55
   mpg=miles/gals
   PRINT"MPG="mpg
   sum=sum+mpg
   count%=count%+1
   FORMAT,2,1
   VDU2:PRINTTRIM$STR$miles+" "+TRIM$STR$litres+SPC(3);
   PRINTTRIM$STR$mpg:VDU3
.hop
UNTIL miles=0
VDU2
PRINT"Average MPG= "sum/count%
VDU3
ON ERROR OFF
RETURN
```

Normally the program will print figures to seven decimal places. Since such accuracy is ludicrous for our purpose we can use the FORMAT command to alter the display to just one decimal place. This has no affect on the accuracy of the calculation.

Since numbers are printed with leading spaces, these spaces are removed by converting each number to a string of characters (STR$) and trimming off the spaces (TRIM$). Try removing TRIM$ to see its effect.

It is important to avoid using CLS after VDU2 since this statement clears the screen by sending a stream of line feeds. Sending line feeds to your printer is a good way of using lots of paper!

Sub Procedures

Sub procedures can be used as a means of splitting the program into readable sections or to reduce duplication of routines.
Sub procedures must always begin with a label preceded by a full stop.
A label can be any combination of characters but not a keyword listed in the reference section.
The sub procedures can come after the main body of the program or may precede the .START label, or may be mixed:

By convention, however, sub procedures usually follow the main program. A program may be shortened by the use of a sub procedure which is used more than once, as shown in the next example:

Example11

```
.START
PROClookup
PRINT"The date is ";date$
INPUTLINE"Type an interesting sentence: ";L$
PROClookup
PRINT"Your sentence was:-"'L$
PROClookup
END
.lookup
date$=ITEM$(TIME$,1,".")
date$=ITEM$(date$,2)
time$=ITEM$(TIME$,2,".")
date@=@date$
date$=STR$date@
PRINT'"The time is ";time$
RETURN
```

```
The time is 01:12:46
The date is 17th May 1989
Type an interesting sentence: Here is my sentence xxxx.

The time is 01:13:12
Your sentence was:-
Here is my sentence xxxx.

The time is 01:13:14
```

This example uses the calendar/clock found in the BBC Master which returns TIME$ in the format: "Wed,17 May 1989.01:12:46".
If the current time is to be found several times during the running of a program then the use of such a Procedure will reduce the size of the program.

Note:
If you are not using a BBC Master you can still try the example by changing TIME$ to XTIME$ and typing -

XTIME$="Wed,17 May 1989.01:12:46"

- before running the program. Of course the time and date will not change!

Dates

The previous example program makes use of the special symbol "@" which defines the variable as a date.
Date variables are translated by IBPL in a way which is defined by the FORMAT command.

Example12

```
.START
date@=@"7.10.89"
FORMAT64:PRINT date@
FORMAT65:PRINT date@
FORMAT66:PRINT date@
FORMAT67:PRINT date@
FORMAT1:PRINT date@
FORMAT0:PRINT date@
RETURN
```

```
07/10/1989
  07/10/89
07/OCT/1989
 07/OCT/89
7th October '89
7th October 1989
```

This example shows that to translate a string into a date we need simply to put the "@" symbol in front of it.

The separator is of no consequence and the date string could have been written as "7/10/89", "7 10 89", "7z10z89" or any other single character.

Example13

```
.START
C$="6*5+89"
D@=@C$
PRINT D@
RETURN

6th May 1989
```

We can also perform calculations with dates:

Example14

```
.START
C$="6/5/89"
D@=@C$
PRINT D@-7
RETURN

29th April 1989
```

Example15

```
.START
C$="6*5+89"
D@=@C$
PRINT D@+30
RETURN

29th April 1989
```

Example16

```
.START
C$="6/May/89"
E$="6/JUNE/89"
C@=@C$
E@=@E$
PRINT E@-C@;" days"
RETURN

31 days
```

Commands and Functions

In the foregoing examples you have seen several keywords such as GOTO, PRINT and INPUT. Keywords can be commands or functions or both, depending on their use.

PRINT, for instance, is very evidently a command whereas GET is actually a function because it returns a value. In general, a keyword which is a function will appear on the right hand side of an = sign.

```
A=GET
PRINT A
```

Going it alone

This chapter should have given you sufficient understanding of IBPL programming techniques to allow you to write your own simple programs.

Begin by modifying the examples already explained by introducing different keywords from the Reference Section. The Reference section also contains many more examples which you can try.

Once you have gained reasonable confidence in handling numbers and strings it will be time for you to progress to the next chapter which deals with database and index files.

Simple database programs in IBPL

The main purpose of IBPL is to simplify disc file handling. In other words it will allow you to write a simple program to save information to disc and to load it back from disc. Furthermore, you will be able to sort some of this information alphabetically or numerically and to search for certain items of data in the disc file. Information stored on disc in this way is called a "Database file". A disc file which contains information about the alphabetical or numerical order of the database records is called an "Index file".

The best way for you to learn to use IBPL is to USE it. Reading about it will simply bore you and you will not assimilate the information.

A database is a program which handles files for holding information on disc. *Since the creation of a database is a little complex to begin with, a database and index have been provided on the example disc.*

The database called "MYDATAB" has fourteen fields:

The first field is an integer field called "numb" and will be used to hold a consecutive number for each "record card".
The second field is a string field called "name".
The third field is a multiple string field, called "addr", with seven lines and will hold an address and postcode.
The fourth field is a string field called "cat" (category).
The fifth field is a string field called "tel".
The sixth field is a string field called "office"
The seventh field is a string field called "anniversary"
Fields 8 to 13 are string fields called "birthday".
Field 14 is a multiple string field, called "notes", with two lines.

The index called "MYINDX" is based upon the first eight characters of the field "name".

The name should always be of the form -

Simpson,Mr P.B.
or
Simpson,Philip

- so the alphabetical index is based upon the Surname.

The two files, "MYDATAB" and "MYINDX", are used many times throughout the reference section of this book.

The following examples are all contained in the separately available example disc in a file named "PROGS". This file should be loaded into P$ in INTER-BASE 0. Delete from P$ those programs preceding the one you wish to try.

Reading files

Load "PROGS" from the example disc if you have one:-

Example: prog1

```
.START
REMOVE array()
DIM array(),14
READ DB"MYDATAB"
READ REC array()
PRINT array()
CLOSE"MYDATAB"
RETURN
```

Now run the program by typing P <RETURN>

Explanation:
We called the array "array()" for clarity but it could equally have some other name such as "R()" or "FreddyFrog()".

The program REMOVEs the array in case it already exists and redefines it as having 14 fields. The types of fields are not defined but will become defined the first time the array is used.

An "array", by the way, may be thought of as a chest of drawers. Each drawer or "field" can hold a different type of item. In the case of IBPL the choice is: Integer number, Real number (decimal point), String, Multiple line string or Date.

The program then opens the database file on disc for READ only.
The first record found at the start of the file is read into the array. (The "chest of drawers" is now full).
The contents of the array are printed on the screen.
Note that only the multiple string arrays contain carriage returns.
The database disc file is closed.

Suppose that we wanted to read the third record in the database:

Example

```
.START
REMOVE array()
DIM array(),14
READ DB"MYDATAB"
SKIP2
READ REC array()
PRINT array()
CLOSE"MYDATAB"
RETURN
```

By using "SKIP2" we move to the third record and read that.

Try it! We can also print the contents of individual "drawers" or fields. For instance PRINT array(2) would cause the contents of the field called "name" to appear on the screen.

Try substituting PRINT array() with

```
PRINT array(2)
PRINT array(3)
PRINT array(1)
```

In addition you will find the string-handling abilities of IBPL useful in dealing with array(3) since this is a multiple string field. In other words it holds several lines, each separated from the next by a carriage return.

Example: prog2

```
.START
REMOVE array()
DIM array(),14
READ DB"MYDATAB"
SKIP2
READ REC array()
addr$=array(3)
FOR X%=7 TO 1 STEP -1
   PRINT LINE$(addr$,X%)
NEXT
CLOSE"MYDATAB"
RETURN
```

This little routine prints the lines of the address in reverse order! (If you put a semi-colon at the end of PRINT LINE$(addr$,X%); to suppress the carriage return then all the lines will be printed one after the other without a line feed between them. Try it.)

The next example makes use of the alphabetical index.

Example: prog3

```
.START
REMOVE array()
DIM array(),14
READ DB"MYDATAB"VIA"MYINDX"
READ REC array()
PRINT array()
CLOSE"MYDATAB"
CLOSE"MYINDX"
RETURN
```

This example prints the first record which the alphabetical index indicates. That will not necessarily be the first record in the actual database file on disc!
The following modification will print ALL the records listed in the index file. It uses a conditional loop "WHILE NOT END" together with "SKIP" so that each record is read in turn. In this instance the keyword "END" refers to the end of the database file which is one SKIP *after* the last record.

Example: prog4

```
.START
REMOVE array()
DIM array(),14
READ DB"MYDATAB"VIA"MYINDX"
WHILE NOT END
   READ REC array()
   PRINT array()
   SKIP
ENDWHILE
CLOSE"MYDATAB"
CLOSE"MYINDX"
RETURN
```

If your computer has sideways ram available you might like to try the modified program as follows:-

Example: prog5

```
.START
REMOVE array()
DIM array(),14
READ DB"MYDATAB"VIA"MYINDX"
 SELECT RAM ALL
 RESERVE "MYDATAB",13000
 RESERVE "MYINDX",2000
WHILE NOT END
   READ REC array()
   PRINT array()
   SKIP
ENDWHILE
CLOSE"MYDATAB"
CLOSE"MYINDX"
END
```

In this example we make use of the sideways RAM as a buffer memory for some of the records and index keys. There is an initial delay while these files are loaded into RAM from disc but thereafter the program runs considerably faster. The speed increase results from the fact that the program is now reading the file copies from RAM instead of from disc.

Of course, if the database file is large, there will be a pause each time the end of the RAM file is reached since more data will have to be transferred from disc to RAM. If, however, your database is small, or you are searching through only a small portion of it, then the increase in speed is very useful! In addition, the BBC Master has several RAM banks available so by reserving all the available RAM you can handle large "chunks" of data.

Note that the amount of RAM reserved for the index is typically only a tenth of that required for the database since each record is much longer than its respective index key.

Since the records will be printed as fast as the program can find them you might like to add the following after "SKIP":

```
PRINT'"Press a key"
G=GET
CLS
```

Having played with these examples you will appreciate that to READ a record is quite simple!

Writing files

To WRITE a record requires a little more planning because, as in any operation which WRITEs to disc files, there is a possibility of losing data.

Each record in the database file takes up a certain amount of space. We call this space the Record Length. There is usually some spare length in each record which is not used. For this reason, it is possible to alter a record - perhaps to add a postcode or to change a name - and to save it back in the same position on disc even though it is now longer. If, however, it is too long to fit back an error message will be given. It is not possible to corrupt the data, therefore, but once you have modified a record and saved it back on disc the original version is over-written and lost forever.

Example: prog6

```
.START
REMOVE array()
DIM array(),14
USE DB"MYDATAB"VIA"MYINDX"
READ REC array()
PRINT array(3)
```

```
INPUT"New Postcode: "post$
PRINT"Which line of address ";
REPEAT
   X$=GET$
   PRINT X$
   X%=VAL X$
UNTIL X%>0 AND X%<8
addr$=array(3)
LINE$(addr$,X%)=post$
array(3)=addr$
WRITE REC array()
CLOSE"MYDATAB"
CLOSE"MYINDX"
PRINT array(3)
END
```

In this example we allow the postcode to be added or changed. Since it is not readily possible to know which line holds the postcode the program asks for this information. The new postcode replaces the old in the record array, which is then written back to disc.

If we allow alteration of the name there is a different problem. The index is derived from the first eight characters of each name (in lower case letters). When a name is changed, therefore, we must not only write the changed record back to disc but also modify the index to ensure that it is still in alphabetical order.

The following program is simplified by the assumption that the record will still fit back in the same position in the database file on disc. If that were not so then it would be necessary to MARK the record as deleted then to APPEND the longer version to the end of the database file.

Example: prog7

```
.START
REMOVE array()
DIM array(),14
USE DB"MYDATAB"VIA"MYINDX"
WRITE INDEX"MYINDX"
READ REC array()
PRINT array()
INPUT"New name: "name$
oldname$=array(2)
```

```
array(2)=LOWER$oldname$
UNSORT REC array()
array(2)=name$
WRITE REC array()
array(2)=LOWER$name$
SORT REC array()
CLOSE"MYDATAB"
CLOSE"MYINDX"
PRINT array()
END
```

The old record key must be removed from the index by UNSORTing it. The name in the array is changed to lower case because, by convention, the index uses only lower case alphabet.

The modified record is written to the database in the same position. The name field must be changed to lower case before SORTing can be carried out. SORTing adds the name key back to the index file in the correct position. Notice that we must USE the database VIA the index in order to link the two but, because in addition we need to WRITE to the index during the SORTing operation later, we must open the index for WRITE.

Any READ operation on the database, therefore, works by reading the alphabetic listing of the index keys in MYINDX (each key includes a record pointer number) then reading the record in MYDATAB specified by the pointer. The SORT operation which is, by definition, a WRITE TO INDEX operation will work only on the index file. WRITE operations on the database have no affect on the index since they operate on the record indicated by the file pointer which is set by the previous READ operation.

Confused? There are two files on disc: the database file contains only records whereas the index file contains an ordered list comprising only a KEY and a pointer for each record. The pointer is a number which indicates the position of the first character of the record in the database file. The key is (in the case of a string) the first few letters from the record field on which the index is based; in the case of our examples, the key is based upon the first eight letters of the name field.

Since the index KEY is an important concept it is recommended that you *review the Reference section notes* for keywords CREATE INDEX, CREATE USER INDEX, CRITERIA, FIND, KEY$, READ KEY and SORT before continuing.

Displaying, Printing and Editing records

A major feature of any database is its ability to provide facilities for allowing the user to see each record and to edit if necessary.
Before displaying the actual record, however, we usually need to know what each line of data represents.

The following example shows how side headings can be displayed on the screen and edited as necessary. Each user of the associated database program can choose his own unique set of headings, even though the database structure itself is unchanged.

Example: "HEAD"

```
.START
LOCAL H$,OH$,boot$,boot1$,mem$
heading$="Home_tel.|MOffice_tel.|MAnniversary|MBirthday|MBirthday
|MBirthday|MBirthday|MBirthday|MBirthday|MNotes:"
REM Example "HEAD"
CLS
PROCsetupstrings
PROCdisplayheads
RETURN

.displayheads
G$="E"
PROCheadings
REPEAT
   answer$=""
   ESCAPE OFF
      VDU23;8202;0;0;0;
      REM cursor off
      PROClines
      Y$=CHR$131
      *FX4,1
      REM cursor keys off
      *FX21
      VDU23;29194;0;0;0;
      REM cursor on
      *FX4
      REM cursor keys on
      PROCeditheads
      IF ASCG$=27 THEN G$="E"
   UNTIL G$="E"
   ESCAPE ON
ENDPROC

.editheads
LOCAL alt%
point%=1
alt%=0
answer$="N"
PRINTTAB(0,2)CHR$131"EDIT"CHR$134"Press <ESCAPE> when finished "
PRINTTAB(0,16)CHR$134"Edit side headings";
PROClookupdata
IF alt%=1
   PRINTTAB(0,2)CHR$131"Save alterations Y/N               ";
   *FX21
   answer$=UPPER$GET$
ENDIF
PRINTTAB(0,2)STRING$(39," ")
   IF answer$="Y"
      PRINTTAB(0,2)CHR$134"Saving on disc......."
   SAVE heading$,"HEADING"
   ENDIF
ENDPROC answer$

.headings
point%=1
   REPEAT
      set$=LINE$(dat$,point%)
      scrline%=VALITEM$(set$,1)
      title$=ITEM$(set$,3)
      PRINTTAB(0,scrline%) title$+Y$;
      point%=point%+1
   UNTIL point%>10
ENDPROC

.lines
point%=1
fill$="."
  REPEAT
    set$=LINE$(dat$,point%)
    scrline%=VALITEM$(set$,1)
    maxlen%=VALITEM$(set$,4)
    item%=VALITEM$(set$,2)
    title$=ITEM$(set$,3)
    tab%=1+LENtitle$
    PRINTTAB(tab%,scrline%)ITEM$(heading$,point%,CHR$13);
    PRINTCHR$134;
    PRINTSTRING$(maxlen%-1-(LENITEM$(heading$,point%,
        CHR$13)),fill$);"<";
    point%=point%+1
  UNTIL point%>10
ENDPROC

.lookupdata
REPEAT
   set$=LINE$(dat$,point%)
   scrline%=VALITEM$(set$,1)
   maxlen%=VALITEM$(set$,4)
   item%=VALITEM$(set$,2)
   title$=ITEM$(set$,3)
   tab%=1+LENtitle$
   TAB tab%,scrline%
   H$=ITEM$(heading$,point%,CHR$13)
   OH$=H$
   cur%=EDIT LINE (H$,maxlen%,1,32,122)
   ITEM$(heading$,point%,CHR$13)=H$
   IF %C=175
      PROCcursup
      ELSE PROCcursdown
   ENDIF
   IF OH$<>H$ THEN alt%=1
UNTIL %C=27
ENDPROC alt%
```

```
.cursup
PRINTTAB(tab%,scrline%)ITEM$(heading$,point%,CHR$13);
PRINTCHR$134STRING$(maxlen%-1-
(LENITEM$(heading$,point%,CHR$13)),fill$);"<";
point%=point%-1
IF point%<1 THEN point%=10
ENDPROC point%

.cursdown
PRINTTAB(tab%,scrline%)ITEM$(heading$,point%,CHR$13);
PRINTCHR$134;
PRINTSTRING$(maxlen%-1-
(LENITEM$(heading$,point%,CHR$13)),fill$);"<";
point%=point%+1
IF point%>10 THEN point%=1
ENDPROC point%

.setupstrings
Y$=CHR$131
dat$="4,1, 1,13|M5,1, 2,13|M6,1, 3,13|M7,2, 4,13|M8,3, 5,13|M9,4,
6,13|M10,5, 7,13|M11,6, 8,13|M12,7, 9,13|M13,1,10,13|M"
ENDPROC
```

The resulting display looks like this:-

```
EDIT Press <ESCAPE> when finished

1 Home_tel. ...<
2 Office_tel. .<
3 Anniversary .<
4 Birthday ....<
5 Birthday ....<
6 Birthday ....<
7 Birthday ....<
8 Birthday ....<
9 Birthday ....<
10 Notes: .....<


Edit side headings
```

This example permits alteration of the headings by means of the normal editing keys.

The actual work is done within the sub procedure .lookupdata in line cur%=EDIT LINE (H$,maxlen%,1,32,122)

The command EDIT LINE permits editing until a special key is pressed.

In this example, any key whose ASCII value lies outside the range 32-122 will exit from the line editor and return a value in the variable cur%. The procedure is written so that <TAB>, <RETURN> or <cursor down> keys will move the editor to the next line while <cursor up> will move the editor back to the previous line. The IF statement with point% causes roll-over between lines 1 and 10. Pressing <ESCAPE> causes an exit from the program.

The variable maxlen% ensures that no heading can exceed 13 characters in length.

Since a complete explanation of every line would be very tedious to read (let alone write!) please load in the "HEAD" example from the disc into

HEAD$ then learn how it works by altering values. Type HEAD<RETURN> to run the program.

The example shows how side headings can be displayed and edited.
To display and edit database records is no more difficult.

The following example requires the previous one loaded into HEAD$.

Example: CALLHED

```
.START
HEAD
REM calls heading program
PROCsetupstrings
REMOVE R()
DIM R(),14
USE DB"MYDATAB"VIA"NAMEINDX"
USE UNMARKED
REPEAT
   READ REC R()
   PROCdisplayarray
   PRINTTAB(0,3)"Next record Y/N ?"
   G$=UPPER$GET$
   PRINTTAB(0,3)SPC18
   IF G$="Y"
      SKIP
     ELSE
      CLOSE"MYDATAB"
      CLOSE"NAMEINDX"
   ENDIF
UNTIL G$<>"Y" OR END
RETURN

.displayarray
G$="E"
REPEAT
   answer$=""
   ESCAPE OFF
      VDU23;8202;0;0;0;
      REM cursor off
      PROClines
      Y$=CHR$131
      *FX4,1
      REM cursor keys off
      *FX21
      VDU23;29194;0;0;0;
      REM cursor on
      *FX4
      REM cursor keys on
      PROCeditarray
      IF ASCG$=27 THEN G$="E"
   UNTIL G$="E"
   ESCAPE ON
ENDPROC

.editarray
LOCAL alt%
point%=1:alt%=0
answer$="N"
PRINTTAB(0,2)CHR$131"EDIT"CHR$134"Press <ESCAPE> when finished "
PRINTTAB(0,16)CHR$134"Edit record "CHR$131;R(2);SPC12
PROClookupdata
IF alt%=1
   PRINTTAB(0,2)CHR$131"Save alterations Y/N             ";
   *FX21
   answer$=UPPER$GET$
ENDIF
PRINTTAB(0,2)STRING$(39," ")
   IF answer$="Y"
      PRINTTAB(0,2)CHR$134"Saving on disc......."
      WRITE REC R()
   ENDIF
ENDPROC answer$

.lines
point%=1
   REPEAT
      set$=LINE$(dat$,point%)
      scrline%=VALITEM$(set$,1)
      maxlen%=VALITEM$(set$,4)
      item%=VALITEM$(set$,2)
      title$=ITEM$(set$,3)
      tab%=17
      PRINTTAB(tab%,scrline%);SPC20
      PRINTTAB(tab%,scrline%);R(point%+4)
      point%=point%+1
   UNTIL point%>10
ENDPROC

.lookupdata
REPEAT
   set$=LINE$(dat$,point%)
   scrline%=VALITEM$(set$,1)
   maxlen%=VALITEM$(set$,4)
   item%=VALITEM$(set$,2)
   title$=ITEM$(set$,3)
   tab%=17
   TAB tab%,scrline%
   H$=R(point%+4)
   OH$=H$
   cur%=EDIT LINE (H$,maxlen%,1,32,122)
   R(point%+4)=H$
   IF %C=175
      PROCcursup
      ELSE PROCcursdown
   ENDIF
   IF OH$<>H$ THEN alt%=1
UNTIL %C=27
ENDPROC alt%

.cursup
PRINTTAB(tab%,scrline%)R(point%+4);
point%=point%-1
IF point%<1 THEN point%=10
ENDPROC point%

.cursdown
PRINTTAB(tab%,scrline%)R(point%+4);
point%=point%+1
IF point%>10 THEN point%=1
ENDPROC point%

.setupstrings
Y$=CHR$131
dat$="4,1, 1,13|M5,1, 2,13|M6,1, 3,13|M7,2, 4,13|M8,3, 5,13|M9,4,
6,13|M10,5, 7,13|M11,6, 8,13|M12,7, 9,13|M13,1,10,33|M"
ENDPROC
```

This program initially runs the previous example program (HEAD$) to allow editing of headings, then reads the first database record and displays fields 5 to 14 for editing. The resulting display will look something like this:-

```
  EDIT Press <ESCAPE> when finished

  1 Home_tel. ...<021-545-4389
  2 Office_tel. .<021-455-6323
  3 Anniversary .<05/06/51
  4 Birthday ....<16/07/35
  5 Birthday ....<23/02/32
  6 Birthday ....<30/03/53
  7 Birthday ....<
  8 Birthday ....<
  9 Birthday ....<
 10 Notes: ......<Has Atari ST

 Edit record Addison,J
```

The program has been kept as near as possible to the previous example "HEAD" in style and headings. Indeed, some of the procedures share the same name in each program. While this is not always good practice (because it could be confusing) it has no detrimental effect on the running of either program. Beware, however, of shared variable names which are not declared as LOCAL.
It is possible to combine both programs in order to utilise common subroutines, albeit with modification. This overall reduction in size could be combined with the omission of the heading-edit facility to simplify the program still further.

The programs could be extended to include record fields 1 to 4 (number, name, address and category). At present, the example indicates the name but does not allow it to be edited.

The examples are simplified for ease of understanding and it should be noted that no error checking is used. For instance, it would be possible to increase the size of the record beyond its maximum length. Consequently, the record would not then fit back in the database file. However the necessary MARK, APPEND and SORT routines to cope with this possibility have *not* been included, here.

Making use of dates

Birthdays are recorded as strings but, provided certain recognisable formats are used, could readily be converted to dates. For instance, assuming R(6) contains "16/7/35"

```
bday1@=@R(6)
PRINT bday1@
```

results in:

16th July 1935

You can make this a little more foolproof by trapping possible errors.
Suppose that R(6) contains "17/667"

```
.START
ON ERROR bday1@=@"31/12/99":ON ERROR OFF:GOTO hop
bday1@=@R(6)
ON ERROR OFF
.hop
PRINT bday1@
```

Results in:-

31st December 1999

To understand this chapter fully please read the Reference Section notes about EDIT, EDITLINE, DISPLAY, ON ERROR, SHOW, POS, VPOS, PRINT and TAB before continuing.

More program examples

The following two program examples may be found on the disc.
To print them out for studying please load them into INTER-WORD. To run them, load them into INTER-BASE as described, below.

"CONVERT"

Since it is occasionally useful to transfer a program written in BBC BASIC to INTER-BASE, the "CONVERT" program was developed. It will not cope with assembly language, however! The original BASIC program in memory must be spooled onto disc by typing:

```
*SPOOL prog <RETURN>
LIST <RETURN>
*SPOOL <RETURN>
```

Load the spooled text into INTER-BASE 0 by typing:

```
*IB.PMENU <RETURN>
LOAD prog$,"prog" <RETURN>
```

Type:

```
LOAD CONVERT$,"CONVERT"
```

If the BASIC program is very long it is wise to TOKENISE the CONVERT$ and, if possible, to INSTALL it in ram (as described in the next section).

To run the conversion type:

```
CONVERT <RETURN>
```

The conversion will take a few minutes.

Use INTER-BASE 0 menu option 5 to view prog$.

The program will not be converted fully but you can run it and find where the remaining errors occur. Look especially for DIM, READ, RESTORE and DATA where differences in syntax occur between BASIC and IBPL. In the case of DIM, the array in BASIC is set up typically as DIM array$(4,16) whereas the same array in IBPL might be:

```
REMOVE array()
DIM array(),5
FOR X%=1 TO 5
FOR Y%=1 TO 16
   DIM array(X%),Y%
NEXT
NEXT
```

This is more complex to set up than in BASIC and, whereas in BASIC element zero exists, in INTER-BASE it does not. The lowest order array element is number one.

In the case of READ and DATA, there are large differences between BASIC and IBPL which you should look up in the Reference Section.

"RETRIEV"

The program was developed to retrieve a database which had become corrupted in the middle. This will certainly work with simple corruption such as incorrect overwriting of a record but will not necessarily cope with a magnetically corrupted disc where whole sectors are missing. Nor will it cope with a database whose "header" is corrupted. Always keep a BACKUP copy!

The program will run faster if TOKENISEd.

ROM Programs

This section describes how to put programs into Sideways RAM or more permanently into EPROM.

Important key words

CLEAR RAM, LOAD RAM, INSTALL, SAVE RAM,
RAM SPACE, RAM STATUS, REMOVE, TOKENISE

So far this book has considered only those programs which reside in user memory in a form that can be edited. However, INTER-BASE provides a facility which will increase the running speed of any IBPL program and, if desired, allow one or more programs to be stored in Sideways RAM (SRAM) or in EPROM.

TOKENISE

The command TOKENISE<prog string> will condense the specified program into a format where every keyword is converted into a single-byte code or "token". In fact numbers and other items are also "tokenised" and, once this is done, the program can not be converted back to its readable form. A copy should always be saved on disc before tokenising! When a tokenised program is run an increase in speed of between 2 and 4 times can usually be achieved.

If, during tokenising, an error is found, then this will be announced. However, not all errors will be noticed. Usually, a reference to a non-existent procedure will be reported but the omission of, for instance, ENDIF will not.

Labels are not tokenised, so error reporting will identify the last label encountered. However, there can be no display of the actual faulty program line. In order to locate the exact error line it is often helpful to load the original program back from disc; add lots of labels between the suspect lines; save another copy on disc; tokenise the program again then run it and note which label is reported.

INSTALL

Once a program has been tokenised it can be transferred to Sideways RAM. For instance, if your program is in prog$ then you would install it in SRAM 4 as follows:-

```
TOK.prog$
CLEAR RAM 4
INST.4,prog$
```

There is no need to type each full command since INTER-BASE recognises abbreviations. The commands could also be put into a program string and run so that, if you are repeatedly altering a program and installing it, you need type the command sequence only once.

Although the program has now been installed there remains a tokenised copy in prog$. If you run the program, this copy will be active and the version in SRAM will be ignored. The original must, therefore, be removed.

REMOVE

A strange quirk of INTER-BASE is that if you REMOVE the default string (i.e. the one whose name appears against menu option 5) then the string contents will be removed but the string itself will remain as a null string. Attempting to run it at this stage will produce the error message "No START".

To overcome this problem use menu option 5 to select some unused variable name as the default. Then remove the program string by typing (in this instance) REMOVE prog$ <RETURN>.

Now prog$ will no longer exist in user memory but when you type:

```
prog <RETURN>
```

the program in SRAM will be run.

You may install as many tokenised programs in SRAM as there is room for. Each bank of SRAM will hold programs up to a total of about 15k bytes since roughly 1k byte is occupied by a machine code header.
More than one program can be installed in any SRAM bank. More than one SRAM bank may contain programs but beware of installing programs into any SRAM bank which you have, or will, SELECT as workspace (see SELECT and RESERVE in the reference section).

CLEAR

Once a program has been installed it can not be removed alone. If the SRAM is CLEARed then **all** the programs in that bank will be erased.
Note that the command CLEAR removes the header but does not fill the SRAM with &FF bytes.

SAVE

Programs installed in SRAM can be saved on disc by the SAVE RAM command. In fact you may type SAVE RAM or SAVE ROM - both are equal in effect.

```
SAVE RAM 4,"progrom" <RETURN>
```

LOAD

Programs may be loaded in a similar fashion:

```
LOAD RAM 4,"progrom" <RETURN>
```

You can determine what programs are present in Sideways RAM and ROM by typing the help command:

```
*H.IB. <RETURN>
```

which will list the contents of each bank.

SHOW

The example program "SHOW" is very long and much too complicated to re-type from a listing. It is therefore only available if you have the example disc.
Load the program "SHOW" into _SHOW$. (The preceding underline character is important). It was developed to produce a menu which displays all INTER-BASE programs currently installed in RAM or EPROM, except for those whose names begin with the underline character "_".

Those programs which you will use merely as subroutines to a main program can be made "invisible" in this way.

IMPORTANT: no program name should contain the underline character except as the first character.

Load the program "SHOW" into INTER-WORD. Print it out to see how it is written.

The program uses the variable, U$, to install machine code directly into memory at &B00. This code is a "user printer" routine which transfers the INTER-BASE program names to the screen. The actual operation of the machine code is beyond the scope of this book but the program itself serves as a good example of what is possible and also provides you with a working utility!

Note that U$ is defined as a long string with .START and RETURN so that, by putting U<RETURN> in the main program, we actually call U$ as a program in its own right so it will install the machine code.
This program has been used in both a BBC B and a BBC Master.
When installed in RAM or EPROM "_SHOW" is invisible to itself.
That is to say it will list other IBASE programs which do not begin with the underline character but will not list itself. (Naturally it will also not list itself if it is run in user memory).

Other INTER-BASE programs can be installed in the same RAM bank as "_SHOW" and may subsequently be SAVEd to disc and blown into EPROM.

Booting from disc

It is possible to include a !BOOT program on disc which will load the program _SHOW and run it to produce a menu of other programs.

!BOOT programs can be typed directly into INTER-BASE and saved onto disc.

Example

```
*IB.PMENU
LOAD K$,"SHOW"
RUN K$
```

This can be saved directly as !BOOT

Remember to type *OPT4,3 to allow the boot option to work from disc.

A more sophisticated system would be to program the RAM image containing _SHOW into EPROM and to plug this into an appropriate socket in the computer.

The !BOOT program is then simplified to

```
*IB.PMENU
_SHOW
```

The need for booting from disc can be avoided altogether by constructing an additional EPROM which contains special codes. Unfortunately, these codes must be in an EPROM alone since the format is incompatible with INTER-BASE programs.

The ROM so constructed contains a machine code !BOOT routine which responds when you hold <SHIFT><DELETE> and press <BREAK>.

This !BOOT code programs key 9 with the command "*EXEC BRING".

"BRING" is the actual boot program which enters *IB.PMENU and runs _SHOW.

A ROM image "BOOTrom" with these codes is on the example disc.

It may be loaded into SRAM or be programmed into an EPROM. The IBPL program "_SHOW" must also be present, either as _SHOW$ in INTER-BASE, in RAM or in an INTER-BASE ROM.

Additional Commands

IBPL provides special labels which are recognised only in a program which has been tokenised and installed in RAM (or EPROM).

.ENTRY is a label which designates the starting point of a program when run by the special command ENTER<program name>.

Once a program has been ENTERed in this way it is impossible to exit without again using the command ENTER to enter a different program.

The system is suitable only for serious programmers and will have little interest for most people.

The built-in programs PMENU (INTER-BASE 0) and MENU (Database) use the ENTER system. It is possible to use ENTER PMENU from your own program, for instance, to return to INTER-BASE 0 menu.

There are three more special labels for use in ROM based programs:

.ERROR

If an error occurs and the error trapping system ON ERROR is not in force then program execution will continue from this label.

.FATAL

If a fatal error occurs then the program will resume from this label.

.RESTART

If an END statement is reached then the program will jump to this label.

```
Example
.START
REM this is myprog$ in RAM.or EPROM.
ENTER myprog
REM a program can enter itself.
```

```
.ENTRY
REM this is the entry point.
FATAL
REM if a fatal error occurs, memory could be corrupted, so the
setup must be done again.
PROCsetup
REM set up various strings and arrays for the program to use.
REM normally this needs to be done only once.
.ERROR
REM non-fatal errors such as pressing <ESCAPE> will return here.
.PROCmenu
RETURN
.menu
REM the menu procedure can be here.
ENDPROC
.otherprocs
REM other procedures can be here.
ENDPROC
```

Communicating with INTER-WORD

It is possible to communicate between the INTER- series packages and to call upon INTER-BASE programs from within these packages. Because of the limitations of the BBC Computer memory organisation, however, the communication is not as friendly as we might wish. In fact some of the possibilities are achieved only by downright "dirty" methods.

However, since a lot of users want the ability to import names and addresses or similar information from a database into INTER-WORD, this chapter gives details of the methods available.

It is important to understand that, while several packages can exist in memory, only one INTER- package can be "active" at one time. Communication, therefore, is strictly one-way with the "active" package COPYING data from a "dormant" package whose text or data was compacted into memory when the "active" package took control. The "dormant" package can not change in size, consequently data can be neither added to nor deleted from it, whereas the "active" package has room to expand and contract.

The saving grace is that INTER-BASE programs can be run even when INTER-BASE itself is dormant, provided that some memory is ALLOCated as workspace.

Important Note:
Error trapping is not possible when an IBPL program is run from another INTER- package. The command ON ERROR has no effect.

The following general procedure can be applied to programs running from INTER-WORD.

Set up a !BOOT program on disc which will select IB.PMENU. Load the desired program(s) if not already resident in sideways RAM/EPROM. ALLOCate sufficient memory as workspace. Initialise any variables then call INTER-WORD as the "active" package.

The following short program can be typed into INTER-BASE and saved as !BOOT. Remember to type *OPT4,3 to enable the "EXEC" disc option.

```
*FX210,1            Turn off sound
*IB.PMENU           Enter IBASE
LOAD P$,"IWcom"     Load program
ALLOC 4000          Allocate workspace
D%=0                Initialise variable
*IW.                Enter INTER-WORD
*FX210              Turn on sound
:P                  Run program
```

Delete the !BOOT program from INTER-BASE and load "IWcom". The following example shows some of the possibilities. Note that you can run it without first using the !BOOT program, provided that you carry out the initialisation by ALLOCating workspace, setting D%=0 and then selecting INTER-WORD. The ALLOCation needs to be done only once but D% must be reset before each run.

Example: "IWcom"

```
.START
REM You must set D%=0 before running this program from INTER-
WORD.
REM Not more than 32 characters can be "Stuffed" at a time.
REM this program MUST be in P$
PROCdefkeys
EXPORT"Hi There"
CASE D%
  WHEN 0
    V$="K"+ES$+Cf4$+AU$+"67"+ES$+Cf6$+"27G"+AD$+"27H"
    V$=+ES$+ES$+":P"+RE$
    EXPORT"Hello. This is a line of text"
  WHEN 1
    V$="K"+ES$+Cf5$+AD$+AR$+AD$+AD$+"3"+ES$+ES$+":P"+RE$
    EXPORT"This is another"
  WHEN 2
    V$="K"+ES$+Cf5$+AU$+AU$+AU$+AR$+AR$+DEL$+"\"
    EXPORT"And another"
ENDCASE
D%=D%+1
PROCbuffstuff V$
RETURN

.buffstuff ^V$
LOCAL X%
FOR X%=1 TO LENV$
A$=STR$ASCV$[X%]
OSCLI"FX138,0,"+A$
NEXT
ENDPROC

.defkeys
Cf4$="|!$"
Cf5$="|!%"
Cf6$="|!&"
AR$="|!|M":REM right
AD$="|!|N":REM down
AU$="|!|O":REM up
DEL$="|?":REM delete left
RE$="|M":REM return key
ES$="|[":REM escape key
ENDPROC
```

At first sight the program might look a little complicated but let's take it step by step.

In order to set up various menu options in INTER-WORD it is necessary to press several function keys and cursor keys. INTER-BASE has no fingers, consequently we must simulate the key presses by putting appropriate codes directly into the keyboard buffer. This method is considered "dirty" and would not usually be recommended for serious programming. However, it represents the only way in which INTER-WORD options can be controlled. It is, however, effective and can give reliable, consistent results, provided that you understand how it works. Anyway, it's good fun!

"PROCdefkeys" defines the codes needed to simulate the keys we shall be using. This action is not strictly necessary - we could use the codes directly - however it makes the program more understandable to use, for instance, ES$+Cf4$ to represent <ESCAPE> <CTRL>+<f4> than to use |[|!$ which is the actual code! A complete list of codes is given at the end of this chapter.

The command EXPORT transfers the text - Hi There - into the INTER-WORD text editor at the cursor position.

Since D% at this point is zero the CASE statement goes to WHEN 0 and constructs V$ from the required codes.

The command EXPORT transfers the text - Hello. This is a line of text - into the INTER-WORD text editor at the cursor position.

Since D% is still zero the program passes to ENDCASE then increments D%.

The memory location of V$ is passed to the procedure PROCbuffstuff which puts the string of codes into the keyboard buffer. When the RETURN statement is reached the program returns control to INTER-WORD which responds with "Press any key" and looks at the keyboard buffer. The first character in V$, therefore, must simulate a key press and the program uses "K", (although any letter would do). The next action must be to press <ESCAPE> from the menu in order to enter the text editor and this is achieved with the code |[(stored in ES$).

The next codes effectively press <CTRL><f4>, <cursor up>, type the footer position as line 67 then <ESCAPE> back to the text editor. Press <CTRL><f6>, type 27G (27,"G" to set emphasised text), <cursor down>, type 27H, <ESCAPE> to text editor, <ESCAPE> to INTER-WORD menu then run the program again by entering :P<RETURN>.

The command EXPORT transfers the text - Hi There - into the INTER-WORD text editor at the cursor position.

This time the program is run with D%=1 so the second version of V$ is constructed.

The command EXPORT transfers the text - This is another - into the INTER-WORD text editor at the cursor position.

D% is incremented to 2 and V$ is put into the keyboard buffer.

Control is returned to INTER-WORD which accepts the simulated key presses as follows:

Press any key "K", <ESCAPE> to text editor, <CTRL>+<f5>, <cursor down>, <cursor right> to change Page range to "some", <cursor down> twice and type "3" to select the last page to be printed. <ESCAPE> twice to get back to INTER-WORD menu then run P$ again.

With D% = 2 the final string of codes is entered to change the pad character to a backslash. This time the program is not RUN again so control remains with INTER-WORD. Press <ESCAPE>.

The reason that we can not put the control codes into the buffer in one long V$ is that the keyboard buffer can accept no more than 32 characters at a time. The program is written, therefore, in a recursive pattern so that this restriction is overcome. This recursion can create real headaches in a larger program and is best avoided if at all possible by keeping the number of codes to a minimum. (It is more sensible to load a "blank page" from disc with the options already set up!)

Further notes

If an inverted comma represents a code it must be preceded by a double bar character, thus |", otherwise INTER-BASE interprets it as the start of a string.

You can set up a function key to run an INTER-BASE program by typing, for instance, *FKEY0 :Prog|M <RETURN>. This definition could be included in the !BOOT file.

Communicating with INTER-CHART

The !BOOT program is similar to that used previously:

```
*KEY0 :D%=0|M:P|M     Define key f0
*FX210,1              Turn off sound
*IB.PMENU             Enter IBASE
LOAD P$,"ICcom"       Load program
ALLOC 4000            Allocate workspace
*IC.                  Enter INTER-CHART
*FX210                Turn on sound
*FX138,0,128          Run program (simulate key f0)
```

This time we have programmed f0 to reset D% and run the program. Just as an example the last command *FX138 acts upon key f0 instead of running the program by using :P directly. In key definitions the code |M must be used directly. Don't try to use RE$ - it won't work!

The following program example will draw a labelled histogram in INTER-CHART. There are some important differences between INTER-WORD and INTER-CHART which should be recognised. After the initial RETURN to INTER-CHART there is no need to "Press any key", consequently the "K" is omitted from V$ on subsequent re-entries.

After any simulated function key or cursor key press the keyboard buffer is cleared of further instructions. Such a key press must, therefore, be the last one before control is returned to INTER-CHART. This fact greatly restricts the use of both INTER-CHART and INTER-SHEET since cursor operations and function keys effectively can not be used.

Example: "ICcom"

```
.START
REM You must set D%=0 before running
REM this program from INTER-CHART.
REM The program MUST be in P$
PROCdefkeys
REM Not more than 32 characters can
REM be "Stuffed" at a time.
REM Replace P with the program name.
CASE D%
   WHEN 0
      V$="K"+ES$+"GRAPH1"+RE$+"L"+ES$+":P"+RE$
   WHEN 1
      V$=ES$+":P"+RE$
      FOR X%=1 TO 5
         EXPORT "|"label"+STR$X%
         EXPORT X%
      NEXT
      FOR X%=4 TO 1 STEP -1
         EXPORT "|"name"+STR$X%
         EXPORT X%
      NEXT
   WHEN 2
      V$=ES$+"7"+f3$+"anything further is ignored"
ENDCASE
D%=D%+1
PROCbuffstuff V$
RETURN

.buffstuff ^V$
LOCAL X%
FOR X%=1 TO LENV$
A$=STR$ASCV$[X%]
OSCLI"FX138,0,"+A$
NEXT
ENDPROC

.defkeys
f3$="|!|C"
RE$="|M":REM return key
ES$="|[":REM escape key
ENDPROC
```

Load ICcom into P$ and type the following:

```
D%=0 <RETURN>
*IC. <RETURN>
P <RETURN>
```

You can reset INTER-WORD -SHEET or -CHART to the original options by typing from the relevant menu:

```
:CANCEL<RETURN>
```

Type `Y` in response to `"Are you sure"` and

```
IW.<RETURN>
or
IS.<RETURN>
or
IC.<RETURN>
```

as appropriate to return to the package which is now cleared and reset.

To edit the program type:

```
:EDIT P$<RETURN>.
```

The following list gives the control codes which simulate each key press and can be sent only via the keyboard buffer. EXPORT can not be used with these codes.

<table>
<tr><td>f0$="|!|@"</td><td>f0 key</td><td></td></tr>
<tr><td>f1$="|!|A"</td><td>f1</td><td></td></tr>
<tr><td>f2$="|!|B"</td><td>f2</td><td></td></tr>
<tr><td>f3$="|!|C"</td><td>f3</td><td></td></tr>
<tr><td>f4$="|!|D"</td><td>f4</td><td></td></tr>
<tr><td>f5$="|!|E"</td><td>f5</td><td></td></tr>
<tr><td>f6$="|!|F"</td><td>f6</td><td></td></tr>
<tr><td>f7$="|!|G"</td><td>f7</td><td></td></tr>
<tr><td>f8$="|!|H"</td><td>f8</td><td>(in IW = Delete Marked Section - CLEARS BUFFER !)</td></tr>
<tr><td>f9$="|!|I"</td><td>f9</td><td></td></tr>
<tr><td>Cf1$="|!!"</td><td>CTRL+f1</td><td></td></tr>
<tr><td>Cf2$="|!|""</td><td>CTRL+f2</td><td></td></tr>
<tr><td>Cf3$="|!#"</td><td>CTRL+f3</td><td></td></tr>
<tr><td>Cf4$="|!$"</td><td>CTRL+f4</td><td></td></tr>
</table>

<table>
<tr><td>Cf5$="|!%"</td><td>CTRL+f5</td></tr>
<tr><td>Cf6$="|!&"</td><td>CTRL+f6</td></tr>
<tr><td>Cf7$="|!'"</td><td>CTRL+f7</td></tr>
<tr><td>Cf8$="|!("</td><td>CTRL+f8</td></tr>
<tr><td>Cf9$="|!)"</td><td>CTRL+f9</td></tr>
<tr><td>Sf0$="|!|P"</td><td>SHIFT+f0</td></tr>
<tr><td>Sf1$="|!|Q"</td><td>SHIFT+f1</td></tr>
<tr><td>Sf2$="|!|R"</td><td>SHIFT+f2</td></tr>
<tr><td>Sf3$="|!|S"</td><td>SHIFT+f3</td></tr>
<tr><td>Sf4$="|!|T"</td><td>SHIFT+f4</td></tr>
<tr><td>Sf5$="|!|U"</td><td>SHIFT+f5</td></tr>
<tr><td>Sf6$="|!|V"</td><td>SHIFT+f6</td></tr>
<tr><td>Sf7$="|!|W"</td><td>SHIFT+f7</td></tr>
<tr><td>Sf8$="|!|X"</td><td>SHIFT+f8</td></tr>
<tr><td>Sf9$="|!|Y"</td><td>SHIFT+f9</td></tr>
<tr><td>AL$="|!|L"</td><td>cursor left</td></tr>
<tr><td>AR$="|!|M"</td><td>cursor right</td></tr>
<tr><td>AD$="|!|N"</td><td>cursor down</td></tr>
<tr><td>AU$="|!|O"</td><td>cursor up</td></tr>
<tr><td>SL$="|!|\"</td><td>SHIFT cursor left</td></tr>
<tr><td>SR$="|!|}"</td><td>SHIFT cursor right</td></tr>
<tr><td>SD$="|!|^"</td><td>SHIFT cursor down -(use |!.|!.|!|L instead) *</td></tr>
<tr><td>SU$="|!|_"</td><td>SHIFT cursor up -(use |!/|!/|!, instead) *</td></tr>
<tr><td>CL$="|!,"</td><td>CTRL cursor left</td></tr>
<tr><td>CR$="|!-"</td><td>CTRL cursor right</td></tr>
<tr><td>CD$="|!."</td><td>CTRL cursor down</td></tr>
<tr><td>CU$="|!/"</td><td>CTRL cursor up</td></tr>
<tr><td>DEL$="|?"</td><td>DELETE left</td></tr>
<tr><td>TA$="|I"</td><td>insert tab (also EXPORT"|I")</td></tr>
<tr><td>RE$="|M"</td><td>RETURN key (also EXPORT"|M")</td></tr>
<tr><td>ES$="|["</td><td>ESCAPE key</td></tr>
</table>

* Note: All SHIFT commands clear the buffer, terminating the sequence, so a recursive return to the program is not then possible. The alternative codes for SD$ and SU$ do not have this effect but can work only within a single page (usually adequate).

EXPORT commands

The following codes do not need to be sent via the keyboard buffer and can be EXPORTed directly to INTER-WORD.

Tab character	"|I" or CHR$9
Carriage return	"|M" or CHR$13
Bold start	"|K" or CHR$11
Bold end	"|S" or CHR$19
Underline start	"|L" or CHR$12
Underline end	"|T" or CHR$20
Dotted start	"|N" or CHR$14
Dotted end	"|U" or CHR$21
Centred start	"|P" or CHR$16
Centred end	"|W" or CHR$23
Right align start	CHR$17 only
Right align end	CHR$24 only
Justified start	CHR$18 only
Justified end	CHR$25 only
Left align	CHR$24+CHR$23 only
Embedded pause	"|D" only

The EXPORT command also sends a carriage return unless you terminate the string to be exported with a semi-colon. For instance, the program

```
.START
EXPORT"|K|P";
EXPORT"Hello ";
EXPORT"there.";
EXPORT"|W|S"
RETURN
```

produces in bold, centred type -

```
          Hello there.
```

Note that the Insert ruler (simulate f2) code can be sent ONLY via the keyboard buffer.

Embedded Commands in INTER-WORD

To place an embedded code, other than pause: simulate f1, simulate cursor movements, enter number if necessary then simulate ESCAPE, (followed by ESCAPE again, program re-entry name and Carriage Return if required).

Colon commands

:CANCEL

Cancels current INTER- package. The command prompts "Are you sure" and removes the package if the letter Y is input, leaving the star prompt "*".

Note :CANCEL leaves two items unchanged in the printer setup menu:-

```
        Send line feeds:
and     Printer type:
```

Example

```
.START
V$=":CAN.|MYIW.0|M"
PROCbuffstuff V$
RETURN

.buffstuff ^V$
LOCAL X%
FOR X%=1 TO LENV$
A$=STR$ASCV$[X%]
OSCLI"FX138,0,"+A$
NEXT
ENDPROC
```

This example cancels the current INTER- package, deleting its contents, and enters IW.0. Note that `CAN.` is an acceptable abbreviation of `CANCEL`.

Assuming that this program resides in P$, therefore, typing `:P<RETURN>` in INTER-WORD 0 menu has the effect of clearing all text and resetting all options to their default status.

:KILL

Similar to `:CANCEL` but removes ALL INTER- packages. Use with care! (Very useful for restoring maximum memory for INTER-BASE use.)

Note Like the :CANCEL command, :KILL leaves two items unchanged in the printer setup menu:-

```
        Send line feeds:
and     Printer type:
```

Transferring text

The following commands assist the transfer of text between packages:

:MOVETOP

Moves the *invisible* ROM-LINK pointer to the top of the text in the specified package so that it exists at a position which is one step BEFORE the first character.

Example

```
:IW.1:MOVET.
```

Note: the ROM-LINK pointer is NOT the visible text cursor. It is moved automatically to the top when a package is entered.

:MOVEAFTER<"string">

Moves the invisible ROM-LINK pointer to the first character after the specified string. Begins the search from the current ROM-LINK pointer position, so it is usually necessary to use :MOVETOP first.

Example

```
:IW.2:MOVEA.
```

:MOVEFORWARD <number>

Moves the ROM-LINK pointer forward by the specified number of characters which may be from 1 to 255. If this number is omitted then 1 is assumed.

Example

```
:IW.0:MOVEF.200
```

:GETCHAR <number>

Copies a specified number of characters which may be from 1 to 255 from the invisible ROM-LINK pointer onwards. If this number is omitted then 1 is assumed.

Example

Type the next line into IW.0

```
Goodbye. For the next three days I shall be away.
```

Run the following program from any package other than IW.0

```
.START
:IW.0:MOVET.
:IW.0:MOVEA."For the next "
IMPORTA$,"IW.0:GETC.20"
PRINT A$
RETURN

three days I shall b
```

:GETMARKED

Copies marked text from the specified package.

Example
```
IMPORT A$,”IW.0:GETM.”
EDIT A$
```

Note: The initial pointer position is unimportant but the invisible ROM-LINK pointer is left immediately after the marked section.

:GETTEXT

Copies all text from the specified package and leaves the invisible ROM-LINK pointer at the end of the text.

Example
```
IMPORT A$,"IW.0:GETT."
MODE3
EDIT A$
```

:GETTEXT(C)

Same as :GETTEXT but transfers all rulers and options as well as text.

:GETTO<string>

Copies all text from the current ROM-LINK pointer position up to (but not including) the specified string. The invisible ROM-LINK pointer is left *after* the specified string.

Example
```
IMPORT A$,”IW.0:GETTO|"||M|"”
```

Calling programs from INTER-WORD

Colon commands can be embedded within the word processor text in order to make use of an IBPL program while a text preview or printout is occurring.

To achieve this object simply press <f1> in INTER-WORD; move the cursor down 3 steps; type in your program name next to the colon and press <ESCAPE>. The embedded command will appear at the present cursor position in the text and the program will be run each time the printout or preview reaches this position.

The most obvious use for this facility is to utilise a mailshot program which will print the same text repeatedly but, for each printout, will insert a different name and address.

Note that before using a program in this way you must first ALLOCate some workspace in INTER-BASE 0, otherwise it will have no room in which to store variables since INTER-WORD will take it all!

Type your mailshot letter into INTER-WORD but omit the recipient's address. Instead, place an embedded command there with the code

```
:mshot
```

The following program works with the example database format.
Since "mshot" is called repeatedly it does not know when to close the database. You must, therefore, type

```
:CLOSEALL<RETURN>
```

when it has finished printing all the mailshots.

The program should be loaded or typed into mshot$ in INTER-BASE and workspace allocated. `ALLOC 1000` will do. The program mshot$ may be tokenised and installed in RAM or EPROM, if desired, to leave memory free.

The "category" of search (field number 4) must be specified in U$. For instance, all those people to whom you send Christmas letters may have "X" in the category field of the database so you type U$="X" before using the program. Any record which does not have X in field 4 will be SKIPped over.

Example: "mshot"

```
.START
REM This is mshot$.
REM search category must be in U$
IF TYPE U$=-1 THEN U$=""
REMOVE R()
READ DB"MYDATAB"VIA"MYINDX"
USE UNMARKED
REPEAT
   READ REC R()
   SKIP
   PROCnameit
   rt$=R(4)
   rs$=R(3)
   IF END THEN rt$=U$
UNTIL U$ IN rt$
EXPORT"|I"+L$
X%=0
REPEAT
   X%=X%+1
   adr$=LINE$(rs$,X%)
   EXPORT"|I"+adr$
UNTIL X%=7
EXPORT ' "Dear "+dear$+","
.end
IF END THEN CLOSEALL
RETURN
.nameit
L$=R(2)
R$=TRIM$L$
sur$=TRIM$ITEM$(R$,1):REM everything prior to comma
chri$=TRIM$ITEM$(R$,2):REM everything after comma
IF chri$<>""
L$=chri$+" "+sur$+","
   IF chri$[1,2] IN "MrDrMsHr"
      dear$=WORD$(chri$,1)+" "+sur$
      ELSE dear$=chri$
   ENDIF
ELSE L$=R$
dear$="Sirs"
ENDIF
ENDPROC
```

Notes:
Sub procedure .nameit looks at the name in field 2 and rearranges it so, for instance, "Simpson,Mr J" becomes "Dear Mr Simpson," and "Nelson,Patricia" becomes "Dear Patricia," in the letter.

Before you print, use <CTRL><f5> Printer setup menu to set the number of copies (= the number of recipients). This number can be smaller than the total and, for the purpose of testing, you can use INTER-WORD menu option

```
7)          Preview text
```

to see what would be printed.

Pressing the space bar during previewing will halt the display and pressing a second time will allow it to continue. The keys <CTRL><SHIFT> held down together will also halt the scrolling display while pressed.

Note that the program will continue to skip to the next address on each printing or preview operation. To return to the first address in the database it is necessary to type:

```
:GO START<RETURN>
```

or

```
:CLOSEALL<RETURN>.
```

You could program function key 9 to do this by typing:

```
*KEY9 :GO START|M <RETURN>
```

Always use `:CLOSEALL` after running the mailshot program, otherwise the database files are left open and can cause problems later.

Final notes about colon commands

From INTER-WORD menu you can assign variables (e.g. `:D%=0`) and print them (e.g. `:PRINT D%` or `:PRINT 3*5`) but you can not print strings (e.g `:PRINTP$` or `:PRINT"hello"` will result in an error message).

Commands such as `:EDIT P$` and `:EXPORT P$` may be used.

You can also use the ROM-LINK colon commands such as

```
:IW.1:GETMARKED.
```

Try typing

```
:HELP<RETURN>.
```

Before running an IBPL program from INTER-WORD menu, always enter INTER-BASE 0 menu first and ALLOCate workspace.

Reference section

Some examples in this section are to be found on the example disc.

ABS

absolute value

This function takes the absolute value of a negative number; i.e. it removes the negative sign from a negative result but leaves a positive result alone.

Examples

```
PRINT(6-9)
                        -3

PRINT ABS(6-9)
                        3

PRINT ABS -89
                        89
```

See SGN

ACS

arc-cosine

This function calculates an angle in radians from a cosine value.

Example
```
PRINT ACS 0.6
0.927295218
```

See ASN, ATN, COS, DEG, EXP, LN, LOG, RAD, SIN, SQR, TAN, PI

ADD@

add date

This function adds a specified number of days, months and years to the given date variable and returns the resulting date.

Syntax
ADD@(<date>,<days%>,<months%>,<years%>)

Example
```
date1$="20/6/51"
date2$=STR$ADD@(@date1$,0,0,38)
PRINT date2$
20th June 1989
```

ADD FIELD

add field to database

This function works on the currently selected database.
It may be used to define fields when a database is first created.
It may be used to add fields to an existing database if room is available.

Syntax
ADD FIELD {INT,STRING,DATE,REAL}

Example
```
ADD FIELD STRING,STRING,INT,DATE,REAL,INT
```

See CREATE DB for more details.

ADVAL

read analogue port

This function returns a digital value from the Analogue Input port.
However, it also has important secondary functions.

There are four analogue input ports which can be selected.
Each has an input voltage range of 0 to 1.8 volts and produces a digital output between 0 and 65520. The digital output increases in steps of 16, however, and is best divided by 16 right from the start.

Example
```
A%=ADVAL(1)/16
PRINT A%
      3097
```

The analogue channels are numbered 1 to 4. It takes 10 milliseconds to return a value from one channel. If all four are in use it will take 40 milliseconds.

ADVAL(0) can be used to give information about the "fire" button inputs on the Analogue port.

Example

```
X%=ADVAL(0)AND3
IF X%=0 THEN PRINT"No button pressed"
IF X%=1 THEN PRINT"Left button pressed"
IF X%=2 THEN PRINT"Right button pressed"
IF X%=3 THEN PRINT"Both buttons pressed"
```

Example

```
X%=ADVAL(0)DIV256
IF X%=0 THEN PRINT"No analogue channel conversion complete."
ELSE PRINT"Channel ";X%;" has just completed conversion."
```

ADVAL can also be used with a negative number to check the status of several internal buffers.

Example

```
.START
X%=ADVAL(-1)
PRINT"There are ";X%;" characters in the key buffer."
X%=ADVAL(-2)
PRINT"There are ";X%;" characters in the RS423 buffer."
X%=ADVAL(-3)
PRINT"There are ";X%;" spaces in the RS423 buffer."
X%=ADVAL(-4)
PRINT"There are ";X%;" spaces in the printer buffer."
X%=ADVAL(-5)
PRINT"There are ";X%;" spaces in SOUND channel 0 buffer."
X%=ADVAL(-6)
PRINT"There are ";X%;" spaces in SOUND channel 1 buffer."
X%=ADVAL(-7)
PRINT"There are ";X%;" spaces in SOUND channel 2 buffer."
X%=ADVAL(-8)
PRINT"There are ";X%;" spaces in SOUND channel 3 buffer."
X%=ADVAL(-9)
PRINT"There are ";X%;" spaces in the SPEECH buffer."
```

Note:
Sampling is faster if some channels are not enabled.
*FX16,0 disables all 4 ADC channels.
*FX16,1 enables channel 1.
*FX16,2 enables channels 1 and 2.
*FX16,3 enables channels 1,2 and 3.
*FX16,4 enables all 4 ADC channels.

ALLOC

allocate ROM-LINK space

This function ALLOCates working space for INTER-BASE programs to use while operating from another INTER- package.

Syntax
ALLOC<bytes>

Example

```
*IB.PMENU
LOAD prog$,"myprog2"
ALLOC 4000
*IW.
:prog
```

The example could be used as a !BOOT program to load "myprog2" into prog$ and run it from INTER-WORD.

See the chapter "Communicating with INTER-WORD".

AND

Logical AND operation

AND performs a bit-wise operation on two numbers, producing a result in which only those bits which are "1" in both numbers remain "1" in the answer. Alternatively, AND can perform a logical operation on statements.

Example

```
PRINT %1110 AND %0111
         6
```

(because 6 in binary is 0110)

Example

```
PRINT 15 AND 1
         1
```

AND can also be used in program statements.

Example

```
A$="Y":B$="S"
IF A$="Y" AND B$="S" THEN PRINT "hello"
hello
```

Example

```
X%=1
REPEAT
            X%=X%+1
            A$=CHR$X%
UNTIL A$="Y" AND X%>100
PRINT X%
        345
```

See OR, EOR, NOT

APPEND REC

add a record to an existing file

This command adds a record to the end of the database file on disc.

Syntax
APPEND REC <record>

Partial example

```
IF REC LEN R() <= MAX REC LEN F()
  WRITE REC R()
 ELSE
  READ REC F()
  UNSORT REC F()
  MARK REC
  APPEND REC R()
  R(2)=LOWER$R(2)
  SORT REC R()
ENDIF
```

ASC

convert to ASCII code

This function returns the ASCII code for the FIRST character in the specified string.

Syntax
=ASC<string>

Example

```
PRINT ASC"hello"
       104
```

(104 is the code for h)

Example

```
yes$="A"
PRINT ASC yes$
        65
```

(the code for A)

Example

```
ESCAPE OFF
REPEAT
            *FX21,0
            G$=GET$
UNTIL ASCG$=27
ESCAPE ON
PRINT"Escape key pressed"
```

See CHR$

ASN

arc-sine

This function calculates an angle in radians from a sine value.

Syntax
=ASN<real>

Example
```
PRINT ASN 0.6
0.643501109
```

See ACN, ATN, COS, DEG, EXP, LN, LOG, RAD, SIN, SQR, TAN, PI

ATN

arc-tangent

Syntax
=ATN<real>
This function calculates an angle in radians from a tangent value.

Example
```
PRINT ATN 0.6
0.5404195
```

See ACN, ASN, COS, DEG, EXP, LN, LOG, RAD, SIN, SQR, TAN, PI

BGET

read byte from file

This function reads a byte from the specified file.

Syntax
BGET<handle>
BGET<filename>

Example

```
.START
OPENIN"MYFILE"
PTR"MYFILE"=2
FORY%=1TO8
   x%=BGET"MYFILE"
   PRINT CHR$x%;
NEXT
CLOSE"MYFILE"
PRINT'"Press any key"
K$=GET$
END
```

```
   CDEFGHIJ
```

Notes:

1 See example in BPUT where "MYFILE" is created.
2 Each byte in the file is "got" as an integer.
3 Pointer PTR is set to zero when the file is closed. If PTR were not specified in this example it would print ABCDEFGH.
4 If reading from an unknown file, take precautions not to PRINT bytes which may affect the screen or printer!

See BPUT, BGET$, CHAN, CLOSE, EXT, EOF, LGET$, OPENIN, OPENOUT, OPENUP, PTR.

BGET$

read characters from file

This function reads the specified number of characters from a file. If the length is not specified then only one character is read. Not more than 255 characters may be read at a time. If there are less than the specified number of characters after the current pointer position then an "End of file" error will occur.

Syntax
BGET$<handle> or BGET$<filename>
BGET$(<handle[,<length>]) or BGET$(<filename[,<length>])

Example

```
.START
OPENIN"MYFILE"
PTR"MYFILE"=2
name$=BGET$("MYFILE",4)
CLOSE "MYFILE"
PRINT name$
END
    CDEF
```

Example

```
.START
file%=OPENIN"MYFILE"
PTR#file%=2
name$=BGET$(#file%,4)
CLOSE #file%
PRINT name$
END
    CDEF
```

Brackets must be used as shown if more than one character is to be read.

See BPUT, BGET, CHAN, CLOSE, EXT, EOF, LGET$, OPENIN, OPENOUT, OPENUP, PTR

BITS

return file information byte

This function returns a byte which gives information regarding the current use of an open file.

Syntax
BITS<handle>
BITS<filename>

Example

```
PRINT BITS"MYPROG"

            File not open
```

Example

```
.START
ONERROR PRINT"File not open.":ONERROR OFF:RETURN
byte%=BITS"MYFILE"
ON ERROR OFF
L%=256
type$=""
FOR X%=7TO0 STEP -1
   L%=L%/2
   IF(byte% AND L%)>0 THEN type$=+STR$(X%)
NEXT
IF "0" IN type$ THEN PRINT"File open for read"
IF "1" IN type$ THEN PRINT"File open for write"
IF "2" IN type$ THEN PRINT"Write database file."
IF "3" IN type$ THEN PRINT"Read database file."
IF "4" IN type$ THEN PRINT"Write index file."
IF "5" IN type$ THEN PRINT"Read index file."
IF "6" IN type$ THEN PRINT"Current index via file."
IF "7" IN type$ THEN PRINT"Current index skip file/index via."
CLOSE"MYFILE"
RETURN
```

Run the program and you will see:

```
            File not open.
```

Type the following, then run the program:

```
OPENUP"MYFILE"<RETURN>
```

You will see:

```
            File open for read
            File open for write
```

Notes:

1 The error trap is needed in case the specified file is not open.

2 The test loop is looking at each of the 8 bits in type% since each bit has a special meaning which you can see in the program.

BPUT

puts a byte in the file

This function sends a single byte or a short string to the specified file.

Syntax

BPUT<handle>,<byte> or BPUT<filename>,<byte>

BPUT<handle>,<sstring> or BPUT<filename>,<sstring>

Example

```
.START
OPENOUT"MYFILE"
FOR Y%=ASC"A" TO ASC"B"
            BPUT"MYFILE",Y%
NEXT
CLOSE"MYFILE"
PRINT"Finished"
RETURN
```

Example

```
.START
OPENOUT"MYFILE"
BPUT"MYFILE","ABCDEFGHIJ"
CLOSE"MYFILE"
RETURN
```

Programs may run quicker if a file handle is defined, instead of using the filename itself. This comment also applies to BGET and similar functions.

Example

```
.START
file%=OPENOUT"MYFILE"
BPUT#file%,"ABCDEFGHIJ"
CLOSE#file%
RETURN
```

NOTES:

1 See example in BGET where "MYFILE" is read.

2 Only integer numbers up to 255 and short strings up to 255 characters in length can be sent. Characters must be converted to ASCII codes before being BPUT as integers.

See BGET, BGET$, CHAN, CLOSE, EXT, EOF, LGET$, OPENIN, OPENOUT, OPENUP, PTR.

BUFLEN

determine free space in a record

This function determines the additional space left at the end of a record when the record is first saved to the database. The record may later be lengthened, if desired, into this space.

Syntax
BUFLEN=<integer>
BUFLEN=BUFLEN+<integer>
<integer>=BUFLEN
PRINT BUFLEN

Example

```
.START
WRITE DB"MYDATAB"
len%=BUFLEN
PRINT"Space = "+TRIM$STR$len%
RETURN
      20
```

Note: The database must be open for writing. BUFLEN will not work on a database which is open only for reading but returns the error message "No write DB open".

See LONGREC, MAX REC LEN, REC LEN

CALL

Run machine code program

This function runs a machine code program which already exists in memory. Since there is virtually no space allocated for user programs the main use of the function is with existing Operating System routines such as OSWRCH.

Syntax
CALL<integer>

Example

```
.START
CLS
PRINT''"Turn printer on and press any key"
K$=GET$
*FX3,10
REM enable printer and disable screen.
V$=CHR$27+"G"+"Hello there"+CHR$27+"H"
FOR D%=1TO LENV$
   C$=V$[D%]
   %A=ASCC$
   CALL&FFEE
NEXT
*FX3,0
REM enable screen.
RETURN
```

Notes:

1 In this example the routine at address &FFEE sends the character to the selected stream (in this case, the printer).

2 %A holds the value to be entered into the accumulator. Do not confuse with BASIC A% which performs the same task! Similarly, the other 6502 registers are assigned %X, %Y. The lowest bit of %C holds the carry flag.

Example "PRINTER"

```
.START
PROCsetup
CLS:P.'''"Turn printer on and press any key":G$=GET$
PROCfirstpage
END
.setup
MC%=&B00 :REM put machine code at memory location &B00
REM 6502 code which sends characters to printer buffer-
$MC%=CHR$&A9+CHR$&0C+CHR$&85+CHR$&71+CHR$&A9+CHR$0+CHR$&85+CHR$&7
0+CHR$&A0+CHR$0+CHR$&B1+CHR$&70+CHR$&20+CHR$&EE+CHR$&FF+CHR$&C8+C
HR$&CC+CHR$&FF+CHR$&0B+CHR$&D0+CHR$&F5+CHR$&60
A=&C00 :REM select memory location &C00 for $A string to use.
ENDPROC

.firstpage
A$="1)...Double-density bit image <ESC+L>"+CHR$27+"L":PROCp A$
A$=CHR$200+CHR$0:PROCp A$
SN$=CHR$34+CHR$80+CHR$138+CHR$0+CHR$143+CHR$0+CHR$138+CHR$80
A$=SN$+CHR$34+CHR$0
FOR L%=1TO20
   PROCp A$
NEXT
ENDPROC
.p^A$
?&BFF=LENA$
$A=A$
*FX3,10
REM output to printer only
CALL&B00
*FX3,0
REM output to screen only
ENDPROC
```

The machine code routine at address &B00 sends the characters at &C00 to the printer. BASIC was used to generate the code, which was then typed as a series of CHR$ so that INTER-BASE could put it at the correct memory location. $A determines the specific memory location of the characters to be printed (&C00 in this example). It should be noted that NO memory location is truly safe and corruption could occur.

See USR

CASE ... ENDCASE

conditionally select action

This command allows the selection of different actions according to the result of an expression.

Syntax

```
CASE<expression>
    WHEN<expression>[,<expression>...]
        <statements>
    WHEN<expression>[,<expression>...]
        <statements>
    OTHERWISE
        <statements>
ENDCASE
```

Example

```
.START
K$=UPPER$GET$
CASE K$
            WHEN "A"
                        PRINT"Choice 1"
            WHEN "B"
                        PRINT"Choice 2"
            OTHERWISE
                        PRINT"Wrong choice"
ENDCASE
```

Notes:

1 The indentation used is not necessary but improves the readability of a long program.

2 OTHERWISE is optional.

3 WHEN may be used as many times as necessary but the first WHEN must follow CASE with no statement between them.

CHAN

return file handle

This function returns the handle of an open file.

Syntax
<integer variable>=CHAN<handle>
<integer variable>=CHAN<filename>

Example

```
.START
open%=OPENIN"MYFILE"
file%=CHAN"MYFILE"
CLOSE#open%
PRINT file%
RETURN

            57
```

Example

```
.START
OPENIN"MYFILE"
file%=CHAN"MYFILE"
CLOSE"MYFILE"
PRINT file%
RETURN

            57
```

Note: Returns 0 if file does not exist or has not been opened.

See BGET, BGET$, BPUT, CLOSE, EXT, EOF, LGET$, OPENIN, OPENOUT, OPENUP, PTR

CHR$

Convert ASCII code to a character.

Syntax
<sstring>=CHR$<integer>

Example

```
PRINT CHR$65
A
```

See ASC, GET$, EVAL

CLEAR RAM

clear sideways ram

This function clears the specified Sideways Ram so that a program which has been tokenised can be installed there. In addition it can de-select sideways RAM which has previously been selected as workspace (SELECT RAM).

Syntax
CLEAR RAM <integer>

Example

```
.START
CLEAR RAM 0
LOAD prog$,"myprog"
TOK.prog$
INST.0,prog$
RETURN
```

Notes:

CLEAR does not fill the Sideways RAM with &FF but removes the header.

If you intend to download the RAM into an EPROM, therefore, you will program the EPROM with a lot of junk after the actual program, extending the programming time. Consequently, you might like to clear the RAM by, for instance, loading into it a file of &FF from disc before installing a program.

See INSTALL, LOAD RAM, SAVE RAM, SAVE ROM, SELECT RAM, RAM STATUS, ROM STATUS

CLG

clear graphics screen

This function clears the graphics screen in those screen modes which support graphics.

Syntax

CLG

Example

```
MODE2
.START
CLG
MOVE 200,200
PLOT 5,200,800
PLOT 5,800,800
PLOT 5,800,200
PLOT 5,200,200
RETURN
```

See CLS, COLOUR, DRAW, GCOL, MODE, MOVE, PLOT, VDU, WINDOW

CLOSE

close files

This function closes files, databases and indexes.

Syntax
CLOSEALL closes all files.
CLOSE#0 closes all files.
CLOSE<filename> closes specified file only.
CLOSE<#handle> closes specified file only.

Example

```
.START
USE DB"MYDATAB"
READ REC R()
CLOSE"MYDATAB"
RETURN
```

Example

```
.START
open%=OPENIN"MYFILE"
name$=BGET$("MYFILE",25)
CLOSE#open%
PRINT name$
RETURN
```

See BGET, BGET$, BPUT, LGET$, OPENIN, OPENOUT, OPENUP, READ DB, READ INDEX, UPDATE, USE DB, USE INDEX, WRITE DB, WRITE INDEX

CLS

clear text screen

This function clears the text screen.

Syntax
CLS

Example

```
.START
CLS
PRINT''"Press any key"
K$=GET$
CLS
FOR
I%=1TO2:PRINTTAB(11,I%)CHR$131CHR$141"DEMONSTRATION";CHR$156:NEXT
PRINTTAB(5)CHR$134"(C) Computer Concepts 1989"
WINDOW 0,4,38,20
PRINT''"Here are some symbols."
PRINT'"Press any key"
CLS
PRINT STRING$(255,"#")
RETURN
```

See CLG, COLOUR

CODE$

produce a code

This function produces a special code based on a given string.

Syntax
=CODE$<string>

Example

```
.START
P.CODE$ "BLOGGS"
P.CODE$ "BLIG"
```

produces the same code for both

The code will be the same if words sound similar, although spelt differently. CODE$ can be used to search for close matches in a list of words, which may be derived from a database or may exist as a string.

Example

```
.START
READ DB"MYDATAB"VIA"MYINDX"
REPEAT
    READ REC R()
    R$=ITEM$(R(2),1)
    REM separate surname from initials.
    IF CODE$R$=CODE$"PUKERING"
      PRINT R(2)+" matches "+"PUKERING"
    ENDIF
    SKIP
UNTIL END
CLOSEALL
```

```
Pickering,Mr M T matches PUKERING
```

Note that letters may be in either upper or lower case or a mixture.

See FIND, HUNT, MATCH

COLOUR , COLOR

select colour

This statement selects the text colour and the text background colour in all modes.

Syntax
COLOUR <int>

Numbers up to and including 127 define the text foreground colour.
Numbers greater than 127 define the text background colour.

In a two colour mode (MODE 0, 3, 4 and 6) the following apply:

foreground	background	colour
0	128	black
1	129	white

In a four colour mode (MODE 1 and 5) the following apply:

foreground	background	colour
0	128	black
1	129	red
2	130	yellow
3	131	white

In a 16 colour mode (MODE 2) the following apply:

foreground	**background**	**colour**
0	128	black
1	129	red
2	130	green
3	131	yellow
4	132	blue
5	133	magenta
6	134	cyan
7	135	white
8	136	flashing black/white
9	137	flashing red/cyan
10	138	flashing green/magenta
11	139	flashing yellow/blue
12	140	flashing blue/yellow
13	141	flashing magenta/green
14	142	flashing cyan/red
15	143	flashing white/black

The colours listed are the default colours or "logical" colours.
It is possible to obtain different colours by swapping the logical colour with another. To achieve this goal we can use the VDU 19 statement:

Example

```
.START
MODE 5
COLOUR 2
VDU19,2,5,0,0,0
COLOUR 128
VDU 19,128,131,0,0,0
PRINT"Here is some text"
```

The result will be magenta text on a yellow background.

See GCOL, VDU

COND$ / COND

set index sort condition / test condition

COND$ is used to define a condition which can later be tested for matching records using COND. COND$ can also be used as a function to read the current condition from the current index, as defined by the most recent COND$ command.

Syntax

COND$=<string>	: defines the condition
=COND$	: returns the current condition
=COND	: tests the current condition

Example

```
.START
USE DB"MYDATAB"
CREATE INDEX"MYINDX"ON 2;13
REM index key based on first 13 letters in field 2 of database.
USE INDEX"MYINDX"
COND$="ITEM$(LOWER$(R(2)),1)=""JONES"""
READ DB"MYDATAB"
WHILE NOT END
    READ REC R()
    IF COND THEN SORT REC R()
    SKIP
ENDWHILE
CLOSEALL
```

This example program will create a new index based upon all the records beginning with "Jones" in the database.

See CREATE INDEX

COS

calculate cosine

This function calculates the cosine of an angle in radians.

Examples

```
PRINT COS 0.5
0.877582562
```

See ACN, ASN, ATN, DEG, EXP, LN, LOG, RAD, SIN, SQR, TAN, PI

COUNT

count items

This function counts the number of items within a string.

Syntax
COUNT(<string>[,<sstring>])
<sstring> is the separator. If omitted then a comma is assumed.

Example

```
mylist$="APPLE,ORANGE,BANANA,PASSION FRUIT,GRAPE"
PRINT COUNT(mylist$)
PRINT COUNT(mylist$,"A")
PRINT COUNT(mylist$,"|M")
            5
            8
            1
```

See ITEM$, WORD$, LINE$

CREATE DB

create database

This command creates a database with a given filename.

Syntax
CREATE DB<filename>,<length>[,<info length>]

<length> is the initial length of the file (which may extend itself as data is added, provided there is room on the disc).

<info length> is optional and represents the initial size of the information block. It defaults to 1024 bytes if not specified.

Both values must be a multiple of 256.

The following example creates a database called "MYDATAB" with 14 fields to hold names, addresses and other details. Two indexes are created; "MYINDX" and "INTINDX". The following points are worthy of note:

1. An attempt is made to OPENIN the file MYDATAB to see if it exists. This is always worthwhile since it can avoid accidental loss of an existing database!

2. *FX21,0 is used to clear the keyboard buffer before asking for input.

3. It is necessary to write something in the first record of the database and, in this example, you are asked for your name and address.

4. In the case of a multiple string field it is most important to add a carriage return "|M" at the end of each line. If the address field is to be left blank, for instance, you must still WRITE the appropriate number of carriage returns to the field; in this example there are seven lines so seven carriage returns must be included.

5. The WRITE INFO lines are needed ONLY if you wish your database to be compatible with INTER-BASE database.

Field zero needs 0,"14,7,Neil" where 14 is the number of fields and 7 is the number of lines in the multiple string field.

The remaining fields need information as follows:
<name>,<type><,Qty><commas>0,<line>,#
name is the name of the field.
type is a letter representing field type (i,s,m,r or d).
Qty is the number of lines in a multiple string.
commas is a string of commas.
In the case of a string, put 6 commas.
In the case of a multiple string put (Qty-1)*6 commas.
In the case of an integer put 3 commas.
In the case of a real put 4 commas.
In the case of a date put 2 commas.

6. An index created on a string field should use only lower case characters for compatibility with the INTER-BASE database.

Example

```
R(2)=LOWER$R(2):SORT REC R().
```

Read the next example carefully. It shows many features which can not readily be explained out of context.

Example program "CREATE"

```
.START
CLS
PRINT'"Please insert ADFS formatted disc."
PRINT'"Which Drive number ? ";
d$=GET$
PRINT d$
OSCLI"MOUNT"+d$
OSCLI"."
G$="Y"
X%=OPENIN"MYDATAB"
CLOSE#X%
IF X%<>0
   PRINT''"Database already exists.|MContinue? Y/N"
   G$=UPPER$GET$
ENDIF
IF G$<>"Y" THEN GOTO end
```

```
*FX21,0
INPUTLINE''"Type your surname.."sur$
*FX21,0
INPUTLINE"Type your first name.."char$
CLS
PRINT''"Type your full address."'"Finish with <RETURN>."
addr$=""
X%=0
REM get each line of address + carriage return
REPEAT
   X%=X%+1
   INPUTLINE G$;
   addr$=+G$
   addr$=+"|M"
UNTIL X%>6 OR LENG$<3
REM then add carriage returns to make 7
WHILE X%<7
   addr$=+"|M"
   X%=X%+1
ENDWHILE
REMOVE R()
DIM R(),14
R(2)=INITIAL$TRIM$sur$+","+INITIAL$TRIM$char$
R(3)=addr$
R(1)=0
REM assign nul string values to remaining fields
FOR X%=4 TO 14
            R(X%)=""
NEXT
CLS
PRINT''R(2)
PRINT R(3)
PRINT'"Creating Database. Please wait"
CREATE DB"MYDATAB",16000
USE DB"MYDATAB"
ADD FIELD INT,STRING,STRING,STRING,STRING,STRING,STRING,STRING,ST
RING,STRING,STRING,STRING,STRING,STRING
BUFLEN=20
REM the following information is needed only for compatibility
WRITE INFO 0,"14,7,Neil"
WRITE INFO 1,"Index,i,,,0,1,#"
WRITE INFO 2,"Name,s,,,,,,0,2,#"
WRITE INFO 3,"Addr,m,7,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,0,3,#"
WRITE INFO 4,"Category,s,,,,,,0,10,#"
REM the remaining fields are identical
FOR X%=5 TO 14
            Y%=X%+6
            Y$=STR$Y%
            info$="field,s,,,,,,0,"+Y$+",#"
            WRITE INFO X%,info$
NEXT
APPEND REC R()
PRINT'"Creating 1st index"
CREATE INDEX"MYINDX",8000 ON 2
REM the index is created on field 2
USE INDEX"MYINDX"
GO START
READ REC R()
R(2)=LOWER$R(2)
REM the index should use only lower case characters for
compatibility
SORT REC R()
ENABLE INDEX"MYINDX"
CLOSE"MYINDX"
PRINT'"Creating 2nd index"
CREATE INDEX"INTINDX",4000 ON 1
USE INDEX"INTINDX"
GO START
READ REC R()
SORT REC R()
ENABLE INDEX"INTINDX"
CLOSE ALL
.end
CLOSE ALL
PRINT'"Finished"
RETURN
```

See CREATE INDEX

CREATE INDEX

create an index

This function creates an index file using the specified database field.

Syntax
CREATE INDEX<filename>[,<len>]ON{<fieldnum>[;<fieldlength>],}

<len> is the initial length of the file but will be increased automatically, when necessary, if there is room on the disc.
<fieldnum> is the number of the field to be referred to in creating the index file.
<fieldlength> is the number of characters, in the field, which will be used in the index file. Each type of field has a default value:

Integers	first 4 characters	Reals	first 5 characters
Dates	first 3 characters	Strings	first 8 characters

The fieldlength can be specified to be smaller than these values and, in the case of strings, longer.

Example

```
READ DB"MYDATAB"
CREATE INDEX "nameindx",8000 ON 2;10
USE INDEX"nameindx"
GO START
WHILE NOT END
            READ REC R()
            R(2)=LOWER$R(2)
            SORT REC R()
            SKIP
ENDWHILE
ENABLE INDEX"nameindx"
CLOSE ALL
```

The index consists of the first 10 characters (the "KEY") of field 2 in lower case for each record, plus a number which is the "pointer" or location of the start of that record in the disc file. To find a particular record you need to search for only the first ten characters (in lower case, not capitals).

CREATE USER INDEX

create a sorted file

This function creates an index which can stand alone as a sorted database in its own right. No other database need be open.

Syntax
CREATE USER INDEX<filename>[,<len>] ON {<fieldtype>[;<fieldlen>],}

The list of field types can be specified directly in the form:

```
CREATE USER INDEX"mylist" ON STRING,REAL,DATE,INT
```

where the default values for fieldlength will be used.

For both CREATE INDEX and CREATE USER INDEX the default values are as follows:

Integers 4, Reals 5, Dates 3 and Strings 8.

You can specify your own values so that, for instance, the sorted index is based upon only the first 4 characters of a string and the first two characters of an integer number. Each value must be preceded by a semicolon For example:

```
CREATE USER INDEX"mylist",&100 ON INT;2,STRING;4
data$="45|MJohn|M242|Mary|M23|MSara|M34|MFred|M22|MSimon"
USE INDEX"mylist"
READ data$,"|M"
WHILE NOT EOD
            age%=DATA
            name$=DATA$
            SORT KEY[age%,name$]
ENDWHILE
UNREAD
CLOSE"mylist"
END
```

This program will create a small database containing the first 4 characters of each name (the "KEY"), sorted according to the descending order of age (first two digits only).
The data could, of course be loaded or typed directly into data$ in the form:

```
45
John
242
Mary
23
Sara .... etc.
```

See SORT

CRITERIA

return index structure

This function returns the KEY structure of an INDEX.

Syntax
=CRITERIA(<handle>[,<int1>[,<int2>]])

Example

```
.START
READ INDEX"ITEST"
N%=CRITERIA("ITEST",0,0)
PRINT"No. of fields = ";N%-1
IF CRITERIA("ITEST",N%,0)=&FFFF THEN PRINT"Database Index"ELSE
PRINT"User Index"
REMOVE R()
DIM R(),6
FOR I%=1 TO N%-1
R(I%)=CRITERIA("ITEST",I%,0)
NEXT
PRINT"      Field No.  Type  Length"
FOR X%=1 TO N%-1
```

```
I%=R(X%)
PRINTI%,CRITERIA("ITEST",X%,1),CRITERIA("ITEST",X%,2)
NEXT
CLOSE "ITEST"
END
```

```
No. of fields = 3
Database Index
     Field No.  Type  Length
        1         4     8
        2         1     4
        3         2     5
```

Explanation

When I1% and I2% = 0 the function returns the quantity of fields upon which the index is structured PLUS ONE (=N%).

When I1%=N% and I2%=0 the function returns a pointer to the database record. If this is &FFFF then the index is a normal Database Index. If not then it is a User Index which, by definition, has no Database.

When I1% is any number between 1 and the total quantity of fields used and when I2% = 0, the function will return the number of the field where I1% represents its order of priority.

To make this clearer, imagine a database with 5 fields with an index based upon a SORT of fields 1, 5 and 3 in that order:

When I1%=1, I2%=0, function returns 1 (field number 1)
When I1%=2, I2%=0, function returns 5 (field number 5)
When I1%=3, I2%=0, function returns 3 (field number 3)

When I2%=1 the function returns the field type number so:

When I1%=1, I2%=1, function returns 1 (Integer field)
When I1%=2, I2%=1, function returns 2 (Real field)
When I1%=3, I2%=1, function returns 4 (String field)

When I2%=2 the function returns the length of the KEY.

(The KEY is that part of the string or number upon which the sort is based. For instance if field 3 contains "Mike Smith" and the KEY length is 8 then the KEY would be "mike smi").

When I1%=1, I2%=2, function returns 4 (Integer KEY length)
When I1%=2, I2%=2, function returns 5 (Real KEY length)
When I1%=3, I2%=2, function returns 8 (String KEY length)

See CREATE INDEX

DATA / DATA$

return data from string

This function returns the next item of data from the string specified by the instruction READ.

Example

```
.START
list$="hello ,there, John"
READ list$
WHILE NOT(EOD)
    wrd$=DATA$ : PRINT wrd$;
ENDWHILE
UNREAD
END
```

```
hello there John
```

Example

```
.START
list$="12,23,34"
READ list$
WHILE NOT(EOD)
    numb%=DATA : PRINT numb%,
ENDWHILE
UNREAD
END
```

```
12         23         34
```

Notes:

1. Unfortunately, if an error occurs while READ is in operation it becomes impossible to UNREAD the data string. In fact the only recourse, should this error occur, is to save the program, switch off, on and reload!
2. In addition it is not possible to set the data pointer but only to reset it to the beginning of the data string by using the command RESTORE (which, however, does not work if the end of the data string has been reached, so always add dummy data at the end!)
3. The following routine has proved to be a versatile substitute for reading data. The data string must be defined BEFORE it can be read. The data is read by means of a function which can return only a string (not an integer or a real number). Where a number is required, therefore, convert the string by means of VAL, as the following example shows.

example of READ DATA substitute using FN.

```
.START
pointer%=0
PROCdefine
FOR X%=1 TO 7
   day$=FNread
   PRINT day$
NEXT
FOR X%=1 TO 9
   num%=VAL(FNread)
   PRINT num%*6
NEXT
REMOVE R()
DIM R(),8
FOR X%=1 TO 8
   R(X%)=FNread
NEXT
PRINT R()
END

.read
pointer%=pointer%+1
rd$=ITEM$(data$,pointer%)
=rd$

.define
```

```
data$=""
data$=+"Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday,
15,23,45,65,"
data$=+"47,78,32,54,23,Jane,Alice,Liz,Kate,Rob,Nev"
ENDPROC
```

Note:
The following lines could also be added to .read

```
IF TYPE pointer%<>1 THEN pointer%=0
```

(in case pointer% was not defined originally)

```
IF pointer%>COUNTdata$ THEN pointer%=0
```

(i.e. RESTORE data pointer)
The sub procedure .read may also end as
`RETURN rd$` or `ENDPROC rd$.`

Another example is given under SOUND.

See COUNT, EOD, FN, READ, RESTORE, UNREAD

DEG

degrees

This function converts angles expressed in radians into degrees.

Example

```
PRINT DEG 0.8
45.8366236
```

See ACN, ASN, ATN, COS, EXP, LN, LOG, RAD, SIN, SQR, TAN, PI

DIM

create array

This function creates in memory an array with a specific name and number of elements. An element can be considered as a numbered location where data can be stored.

Syntax
DIM<array name>,<int>

Example

```
.START
REMOVE blocks()
DIM blocks(),4
date@=@"20/6/51"
blocks()=[date@,8.95,"This is a string",45]
PRINT blocks()
PRINT
PRINT blocks(3)
PRINT blocks(2)
PRINT blocks(4)
PRINT blocks(1)
END
```

```
20th June 1951      8.95This is a string        45

This is a string
      8.95
        45
20th June 1951
```

Notes:

1. The element numbers begin at 1, unlike BASIC where they begin at 0.
2. An array can not be redefined without first removing it, hence it is safer to REMOVE it before using DIM in order to avoid errors.
3. An array element has a type associated with it. This type is set the first time that data is stored in each element. Once set, the data type

for a particular element can not be changed. If you try, for instance, to store an integer in an element which first held a string then an error will occur. (The function TYPE can be used to determine the type in order to avoid such an error).

4. Any element of an array can itself be an array.

Example

```
DIM blocks(),4
FOR X%=1 TO 4
            DIM blocks(X%),6
NEXT
```

See REMOVE, TYPE

DISABLE INDEX

remove index name

This function removes an index name from the header block of the current WRITE database.

Syntax
DISABLE INDEX<filename>

Example

```
.START
USE DB"MYDATAB"
DISABLE INDEX"MYINDX"
CLOSE "MYDATAB"
END
```

See ENABLE INDEX, INDEX$

DISPLAY

display string

This function will display on screen a string variable without allowing editing. The current text window is cleared before the string is displayed. All cursor controls work as usual.

Syntax
DISPLAY<stringvar>[,<int>[,<int>]]

The first (optional) integer specifies the cursor position counting from the first character of the string variable.
The second (optional) integer selects the key(s) which may be used to cancel the display. If 1 then <RETURN> will exit. If 2 then <SHIFT> plus cursor key will exit but only when the cursor is at its extreme position in the cursor key direction. <ESCAPE> will allow exit in all cases.

On exit, the resident variable %C (not C%) will contain the ASCII code of the key used to exit. This would be 27 for `<ESCAPE>`, 13 for `<RETURN>`, 142 for `<SHIFT>+<cursor down>`, 143 for `<SHIFT>+<cursor up>` 141 for `<SHIFT>+<cursor right>` and 140 for `<SHIFT>+<cursor left>`.

Example

```
.START
P$="This is a string which is here for the purpose of
demonstration."
DISPLAY P$,45,2
PRINT %C
END
```

(you press `<ESCAPE>`)

```
27
```

Syntax
=DISPLAY(<stringvar>[,<int>[,<int>]])

If DISPLAY is used as a function it returns the cursor position on exit.

Example
```
.START
curs%=DISPLAY(P$,45,1)      (note brackets !)
PRINT curs%
END
```

(you move cursor then press `<RETURN>`)

```
56
```

See EDIT, EDITLINE, SHOW

DIV

return integer result of division

This function returns the whole number part of a division, ignoring any remainder.

Example
```
.START
PRINT 14 DIV 5
END
```

```
2
```

(leaving a remainder of 4)
The result is always an integer.

See MOD

DRAW

draw lines

This statement draws lines on the screen (in graphics modes only).
The end of the line must be defined as coordinates X and Y.

Syntax
DRAW X,Y

Example
```
.START
MODE 4
DRAW 950,800
END
```

See CLG, COLOUR, GCOL, MODE, MOVE, PLOT, POINT, VDU

EDIT

display string for editing

This function will display on screen a string variable, allowing editing. The current text window is cleared before the string is displayed. All cursor controls and editing facilities work as usual.

Syntax
EDIT<stringvar>[,<int>[,<int>]]

The first (optional) integer specifies the cursor position counting from the first character of the string variable.
The second (optional) integer selects the key(s) which may be used to cancel the display. If 1 then `<RETURN>` will exit. If 2 then `<SHIFT>` plus

cursor key will exit but only when the cursor is at its extreme position in the cursor key direction. <ESCAPE> will always allow exit.

On exit, the resident variable %C (not C%) will contain the ASCII code of the key used to exit. This would be 27 for <ESCAPE>, 13 for <RETURN>, 142 for <SHIFT>+<cursor down>, 143 for <SHIFT>+<cursor up>, 141 for <SHIFT>+<cursor right> and 140 for <SHIFT>+<cursor left>.

Example

```
EDIT P$,45,2
```

(Press <ESCAPE>)

```
PRINT %C

27
```

Syntax

=EDIT(<stringvar>[,<int>[,<int>]])

If EDIT is used as a function it returns the cursor position on exit.

Example

```
curs%=EDIT(P$,45,1)        (note brackets !)
```

(Perform editing then press <RETURN>)

```
PRINT curs%

56
```

This knowledge of the cursor position is useful if the string must later be re-entered at the same place.

See DISPLAY, EDITLINE, SHOW

EDITLINE

edit line of text

This command may be used to edit a single line of text but is more powerful than EDIT alone since it allows you to control the layout of text on the screen.

Syntax
EDITLINE<stringvar>,<int1>,<int2>,<int3>,<int4>
No brackets!!!

<stringvar> is the string to be edited. It must not contain carriage returns or each successive screen line will overprint the last, making editing impossible.

<int1> is the maximum string length which MUST be specified. An error will be announced if the string is longer than this value, so it is wise to perform a test first if the length is undefined. Provided that the string length is not more than this maximum it will be displayed for editing. Characters may be added during editing but only up to the maximum defined. Additional key presses beyond the maximum are ignored.

<int2> is the optional cursor position, counting from the first character of the string, when editing begins.

<int3>,<int4> define the minimum and maximum ASCII codes of the characters which may be input during editing. Any codes outside the range so defined will exit from the line editor.

Regardless of the codes defined, the normal editing keys may be used. Apart from left and right cursors, these editing keys will exit from the editor if an attempt is made to go beyond the end of the string. The ASCII code of the key used to exit will be held in the integer variable %C.

Normally the command is used in the form of a function which will also return the cursor position on exit.

Syntax
<intvar>=EDITLINE(<stringvar>,<int1>,<int2>,<int3>,<int4>) Note brackets!!!

The first program, below, works rather like the statement

```
INPUT"Filename: ";file$
```

but is more flexible in that it displays the previous filename (if any) and permits editing. The length of the filename is limited to 7 characters but can be extended to 10 for an ADFS system. Note the initial test to see if file$ exists.

Example

```
.START
len%=7
IF NOT(TYPE file$=4) THEN file$="myfile2"
PRINTTAB(0,10)"Filename: ";
EDITLINEfile$,len%,1,ASC"'",ASC"z"
```

Now you could have:

```
SAVE prog$,file$
```

or similar, followed by:

```
PRINT''"Saving prog$ as ";file$
```

assuming that prog$ holds the program or string you want to save. The function will normally be used within a loop so that more than one line can be displayed at a time. This idea is better understood in the form of another example.

Example "DISPL"

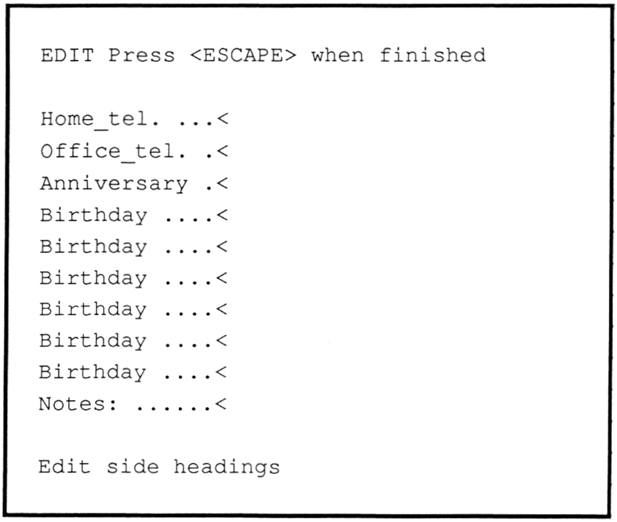
```
.START
head$="Home_tel.|MOffice_tel.|MAnniversary|MBirthday|MBirthday|MB
irthday|MBirthday|MBirthday|MBirthday|MNotes:"
CLS
PROCsetupstrings
PROCdisplayarray
CLS
RETURN

.displayarray
G$="E"
PROCheadings
REPEAT
   answer$=""
   ESCAPE OFF
```

```
      VDU23;8202;0;0;0;
      REM cursor off
      PROClines
      Y$=CHR$131
      VDU23;29194;0;0;0;
      REM cursor on
      PROCeditarray
      IF ASCG$=27 THEN G$="E"
   UNTIL G$="E"
   ESCAPE ON
ENDPROC

.editarray
LOCAL alt%
point%=1
alt%=0
answer$="N"
PRINTTAB(0,2)CHR$131"EDIT"CHR$134"Press <ESCAPE> when finished "
PRINTTAB(0,16)CHR$134"Edit side headings";
PROClookupdata
IF alt%=1
   PRINTTAB(0,2)CHR$131"Save alterations Y/N           ";
   *FX21,0
   answer$=UPPER$GET$
ENDIF
PRINTTAB(0,2)STRING$(39," ")
   IF answer$="Y"
      PRINTTAB(0,2)CHR$134"Saving on disc......."
      SAVE head$,"HEADING"
   ENDIF
ENDPROC answer$

.headings
point%=1
   REPEAT
      set$=LINE$(dat$,point%)
      scrline%=VALITEM$(set$,1)
      title$=ITEM$(set$,3)
      PRINTTAB(0,scrline%) title$+Y$;
      point%=point%+1
   UNTIL point%>10
ENDPROC
```

```
.lines
point%=1
fill$="."
   REPEAT
      set$=LINE$(dat$,point%)
      scrline%=VALITEM$(set$,1)
      maxlen%=VALITEM$(set$,4)
      item%=VALITEM$(set$,2)
      title$=ITEM$(set$,3)
      tab%=1+LENtitle$
      PRINTTAB(tab%,scrline%)ITEM$(head$,point%,CHR$13);
      PRINT CHR$134STRING$(maxlen%-1-
(LENITEM$(head$,point%,CHR$13)),fill$);"<";
      point%=point%+1
   UNTIL point%>10
ENDPROC

.lookupdata
REPEAT
   set$=LINE$(dat$,point%)
   scrline%=VALITEM$(set$,1)
   maxlen%=VALITEM$(set$,4)
   item%=VALITEM$(set$,2)
   title$=ITEM$(set$,3)
   tab%=1+LENtitle$
   TAB tab%,scrline%
   H$=ITEM$(head$,point%,CHR$13)
   OH$=H$
   cur%=EDIT LINE (H$,maxlen%,1,32,122)
   ITEM$(head$,point%,CHR$13)=H$
   IF %C=175
      PROCcursup
      ELSE PROCcursdown
   ENDIF
   IF OH$<>H$ THEN alt%=1
UNTIL %C=27
ENDPROC alt%

.cursup
PRINTTAB(tab%,scrline%)ITEM$(head$,point%,CHR$13);
PRINTCHR$134STRING$(maxlen%-1-
(LENITEM$(head$,point%,CHR$13)),fill$);"<";
```

```
point%=point%-1
IF point%<1 THEN point%=10
ENDPROC point%

.cursdown
PRINTTAB(tab%,scrline%)ITEM$(head$,point%,CHR$13);
PRINTCHR$134STRING$(maxlen%-1-
(LENITEM$(head$,point%,CHR$13)),fill$);"<";
point%=point%+1
IF point%>10 THEN point%=1
ENDPROC point%

.setupstrings
Y$=CHR$131
dat$="4,1,1,13|M5,1,2,13|M6,1,3,13|M7,2,4,13|M8,3,5,13|M9,4,6,13|
M10,5,7,13|M11,6,8,13|M12,7,9,13|M13,1,10,13|M"
ENDPROC
```

The resultant screen should look like the illustration, below, and you will be able to move the cursor and edit the wording:



```
EDIT Press <ESCAPE> when finished

Home_tel. ...<
Office_tel. .<
Anniversary .<
Birthday ....<
Birthday ....<
Birthday ....<
Birthday ....<
Birthday ....<
Birthday ....<
Notes: ......<

Edit side headings
```

See EDIT, DISPLAY, SHOW

ENABLE INDEX

allow database to use index

This command adds the filename of an index to the list of updatable index names which is stored in the information block of the current WRITE database. Your program should be written so that each index listed is kept updated with new entries to your database.

Syntax
ENABLE INDEX<filename>

Example

```
.START
USE DB"MYDATAB"
ENABLE INDEX"NAMEINDX"
CLOSE "MYDATAB"
END
```

See CREATE INDEX, DISABLE INDEX, INDEX$

END

define end of main program

This command prevents the program from running further and will normally return control to the main menu. Procedures are usually placed after this END command. If a program is to be run from within another program or INTER- package then use RETURN or ENDPROC instead of END.

See RESTART, RETURN, ENDPROC

As a function, END has another use.

Example

```
.START
READ DB"MYDATAB" VIA "MYINDX"
USE UNMARKED
GO START
WHILE NOT END
    READ REC F()
    IF "Mr" IN F(2)
       PRINT F()
    ENDIF
    SKIP
ENDWHILE
CLOSE"MYDATAB"
CLOSE"MYINDX"
RETURN
```

In this example the function END causes the WHILE ... ENDWHILE loop to exit when the end of the database file is reached, having printed on screen all those records where "Mr" is found in field 2.

See START

ENDCASE

See CASE ... ENDCASE

ENDIF

See IF ... ENDIF

END PTR

return record pointer

This function returns a number which is a pointer to the byte following the last record (or key) of the current database (or index).

Syntax
=END PTR<handle>

Example

```
.START
READ DB"MYDATAB"
X%=END PTR"MYDATAB"
CLOSE "MYDATAB"
PRINT"End pointer is ";X%
RETURN
```

See EXT, PTR

ENDPROC

See PROC ... ENDPROC

ENDWHILE

See WHILE ... ENDWHILE

ENTER

enter an installed program

This command will run a program which has been tokenised and installed in RAM or ROM.

Syntax
ENTER<program name>

See The chapter "Rom Programs".

ENVELOPE

define sound envelope

Defines a sound envelope in terms of volume and pitch.

Syntax
ENVELOPE,<int1>,<int2>,<int3>,<int4>,<int5>,<int6>,<int7>,
<int8>,<int9>,<int10>,<int11>,<int12>,<int13>,<int14>

A sound envelope has three pitch stages; A, B and C and four amplitude stages; attack, decay, sustain and release. The integer parameters following the ENVELOPE statement have the following functions:

Parameter	Range	Function
int1	1 to 4	Envelope number
int2 bits 0-6	0 to 127	Duration in centiseconds
int2 bit 7	0 or 1	0=auto repeat pitch envelope, 1=no repeat
int3	-128 to 127	Pitch change per step in stage A
int4	-128 to 127	Pitch change per step in stage B
int5	-128 to 127	Pitch change per step in stage C
int6	0 to 255	Number of steps in stage A
int7	0 to 255	Number of steps in stage B
int8	0 to 255	Number of steps in stage C
int9	-127 to 127	Amplitude change per step during attack
int10	-127 to 127	Amplitude change per step during decay
int11	-127 to 0	Amplitude change/step during sustain
int12	-127 to 0	Amplitude change/step during release
int13	0 to 126	Target amplitude at end of attack
int14	0 to 126	Target amplitude at end of decay

Usually only four envelopes may be defined. However, if the statement BPUT# is not used in the program then 16 envelopes may be defined.

Example
```
.START
ENVELOPE 1,129,4,-4,4,10,0,0,-127,-10,-10,-5,126,126
SOUND 1,1,100,200
```

See SOUND

EOD

determine end of data

This function determines when the end of a DATA string has been reached.

Syntax
=EOD

Example

```
.START
mydata$="apple,pear,banana,grapefruit,orange,turnip"
READ mydata$
WHILE NOT(EOD)
            A$=DATA$
            PRINT "Fruit is "+A$
ENDWHILE
UNREAD
END
```

See DATA, DATA$, READ, RESTORE, UNREAD

EOF

end of file

This function detects the end of a file (but not the end of a database). it returns -1 if the end of a file is reached.

Syntax
=EOF<handle>
=EOF#<handle number>

Examples

```
.START
OPENIN"MYFILE"
WHILE NOT(EOF)
             name$=BGET$("MYFILE",10)+"|M"
             PRINT name$
ENDWHILE
CLOSE "MYFILE"
RETURN

.START
N%=OPENIN"MYFILE"
WHILE NOT(EOF)
             name$=BGET$(#N%,10)+"|M"
             PRINT name$
ENDWHILE
CLOSE#N%
RETURN
```

See END, START

EOR

Exclusive OR logic operator

Syntax
<int1>EOR<int2>

If you EOR two binary numbers then the resultant binary number will have a 0 where two 1 digits coincided in the original and also where two 0 digits coincided. It will have a 1 only in those positions where a 1 existed in that position in one binary number alone.

Truth Table for EOR

1 EOR 1 = 0
0 EOR 0 = 0
1 EOR 0 = 1
0 EOR 1 = 1

Example

```
.START
numb1%=%00011111
numb2%=%00101111
res%=numb1% EOR numb2%
PRINT ~res%

30
```

(which is 00110000 in binary)

(The operator % defines the number following to be binary, just as the operator & precedes a hexadecimal number.)

See AND, NOT, OR

ERL$

return label before error

This function returns the label after which the error occurred.

ERM$

return error message

This function returns a message to describe the error.

ERP$

return program name

This function returns the name of the program in which the error occurred.

ERR

return error number

This function returns the error number.

ERR$

return error line string

This function returns the complete line in which the error occurred.

Example

```
.START
REPEAT
   ON ERROR PROCreport
   .flag
   PRINT"Enter function:";
   func$=INPUT$
   y=EVALfunc$
   PRINTy
   ON ERROR OFF
UNTIL y>100
END

.report
X%=ERR
IF X%=17 THEN END
REM <ESCAPE> will exit.
PRINT'"Invalid function"'"Nearest label = ";
PRINT ERL$'"Error number ";ERR;'ERM$'"In program "; ERP$
ENDPROC
```

```
Enter function: 7/0          (you type this)
Invalid function
Nearest label = flag
Error number 70
Division by zero
In program P
Enter function:
```

ESCAPE OFF / ON

disable/enable Escape key

Disables and enables the <ESCAPE> key to prevent inadvertent exit from a program. These commands should be added only after the program is working correctly (or at least only after the program has safely been saved on disc).

Example

```
.START
ESCAPE OFF
REPEAT
   PRINT"Press a key";
   G$=GET$
   PRINT G$
   IF ASCG$=27
      CLS:PRINT''"Escape pressed"
      ESCAPE ON
   ENDIF
UNTIL ASCG$=27
```

EVAL

evaluate an expression

This evaluates an expression string and returns the result.

See example in ERR$

See VAL

EXEC

execute command

This command will execute program commands presented in a string which may not be more than 255 characters long.

```
Example
.START
REPEAT
   PRINT"Type a command"
   com$=INPUT$
   EXEC com$
UNTIL FALSE
```

In practice some fairly rugged error trapping would be needed since ANY command would be accepted, including star commands!

EXP

exponent

This function calculates the value of 2.71828183 raised to any specified power.

Example

```
PRINT EXP 5.6

270.426407
```

See ACN, ASN, ATN, COS, DEG, LN, LOG, RAD, SIN, SQR, TAN, PI

EXPORT

copy string to active package

This command copies a specified string from INTER-BASE to the active INTER- package.

Example
In INTER-BASE PMENU string P$ type the following:

```
This sentence comes from P$
```

Type some text in INTER-WORD and leave the cursor in the middle of the text.

From INTER-WORD menu type the command:

```
:EXPORT P$ <RETURN>
```

You will find your sentence has been copied into INTER-WORD at the cursor position.

You can use the command in a program. Alter the contents of P$ to read:

```
.START
EXPORT"This sentence comes from P"
RETURN
```

From INTER-WORD menu type:

```
:P <RETURN>
```

The program will export its text to the cursor position in INTER-WORD.

See the chapter "Communicating With INTER-WORD".

See ALLOC, IMPORT

EXT

file length

This function reads or sets the length of a file.

Syntax
EXT<handle>=
=EXT<handle>

Example

```
.START
READ DB"MYDATAB"
X%=EXT"MYDATAB"
PRINT"File length is ";X%
CLOSE"MYDATAB"
END

65536
```

Example

```
.START
OPENIN"ITEST"
PRINT EXT"ITEST"
CLOSE"ITEST"
END

1280
```

Example

```
.START
OPENOUT"ITEST"
EXT"ITEST"=1400
CLOSE"ITEST"
```

See END PTR, PTR

FIELDS

return number of fields

This function returns the number of fields per record in the current database.

Syntax
=FIELDS

Example

```
.START
READ DB"MYDATAB"
X%=FIELDS
CLOSE"MYDATAB"
PRINT X%
RETURN

14
```

See ADD FIELD

FILES

return number of open files

This function returns the number of files which currently are open.

Syntax
=FILES

Example

```
.START
READ DB"MYDATAB"VIA"MYINDX"
X%=FILES
CLOSE"MYDATAB"
PRINT X%
X%=FILES
PRINT X%
CLOSE"MYINDX"
X%=FILES
PRINT X%
RETURN
```

The result would be:

```
2
1
0
```

Note that READ DB ... VIA ... opens 2 files!

See CLOSE, OPENIN, OPENOUT, OPENUP, READ, USE, WRITE

FIND

find a key

This command searches the current read index for the specified key.

Syntax
FIND<key string>   (must not be longer than the key in the index).
FIND KEY<key array> (must contain keys in the correct order).
FIND REC<key array> (must contain keys in the corresponding fields).

Example

```
.START
READ DB"MYDATAB"VIA"MYINDX"
PTR"MYDATAB"=0
FIND"fothergi"
REM not more than 8 letters
X%=PTR"MYDATAB"
READ REC R()
Y%=PTR"MYDATAB"
REM CLOSE ALL
PRINT R(2)'R(3)
PRINT "Pointer 1 = ";X%
PRINT "Pointer 2 = ";Y%
```

```
Fothergill,Mr J
Snide & Company Ltd.,
12 Hertford Close,
Emsworth Field,
Luton,
Beds.
LU4 6HJ

Pointer 1 =      0
Pointer 2 =      2416
```

Notes:
The act of FINDing a key in the index does not alter the database file pointer. In order to set the pointer to the record you must first READ the

record from the database. The example demonstrates this. Do not, therefore, FIND a record then attempt to WRITE to it because you will overwrite the first record in the file instead! You must READ first.

The search key may be shorter than the actual key in the index but not longer, otherwise the search will be unsuccessful. (This fact is NOT true in the case of an array). For a key string the default length is eight characters. For an unknown database you can use the function CRITERIA to determine the key length before using FIND.

A search key string must be of the same case as that of the index. It is usual to construct an index with lower case letters only. If you use the wrong case then FIND will be unsuccessful.
Since the search is performed on the index and not on the database itself, the index must be opened for reading.

If the index is based upon more than one field you can use the FIND command with an array containing the relevant keys. For instance, if the index is based upon both the surname and the street number in the example above you could use:

```
FIND KEY["fother",12]
```

If the array already exists as in:

```
z(2)="Fothergill,Mr J"
z(3)="12 Hertford Close, etc...."
```

you could modify it to suit the key structure defined above:

```
.START
REMOVE J()
DIM J(),2
J(1)=LOWER$z(2)[1,8]
REM take the first 8 characters in lower case
J(2)=VALz(3)
REM take just the street number
FIND KEY J()
```

The third format is to use:

```
FIND REC J()
```

In this case you must note that the index was constructed from the first and third fields of the database (Surname and street number). Consequently, the array must contain the key information in its first and third fields. Note that only those fields upon which the index was originally constructed need be filled. The other fields may be undefined or may hold any "garbage".

```
.START
REMOVE J()
DIM J(),3
J(1)=LOWER$z(2)
REM take the surname in lower case
J(3)=VALz(3)
REM take just the street number
FIND REC J()
```

It was necessary to convert the surname to lower case and part of the address string to a number. If the elements of the array already match those of the index, you may be able to use the record array as it stands:

```
FIND REC z()
```

=FIND may also be used as a function: a useful facility if you are not certain that the key exists since an error would result from the use of the command whereas the function will return a value of -1 if TRUE (found) and 0 if FALSE (not found).

Example

```
.START
REMOVE R()
DIM R(),14
READ DB"MYDATAB"VIA"MYINDX"
X%=FIND"fothergi"
IF X%=-1
   READ REC R()
   PRINT R(2)'R(3)
   ELSE PRINT"Not found"
ENDIF
CLOSE ALL
RETURN
```

See CRITERIA, HUNT

FN

return result

Returns the result of a programming action from a sub procedure.

Syntax
=FN<label>[(<value>{[,<value>]})]

Example
```
.START
X%=FNworkitout (8,9)
PRINT X%
END
.workitout x%,y%
= x%*y%

72
```

Instead of the = sign the sub procedure could also end as:
`RETURN x%*y% or ENDPROC x%*y%`

Example
```
.START
X$=FNworkitout ("Hello","there")
PRINT X$
END
.workitout x$,y$
= x$+" you "+y$+" with the hat"
```

The result of this example would be:
```
Hello you there with the hat
```

It is possible to write a procedure in such a way that it can be called both as a function and as a procedure by separate parts of the program.
Where speed is important, for instance where a function is called repeatedly by separate parts of the program, parameters may be passed as variables of differing variable names. The following example passes

the parameters as strings. The program actually copies these and presents them to the function sub-procedure on each loop.

Example

```
.START
X%=0
TIME=0
REPEAT
   X$=FNworkitout ("Hello","there")
   PRINT X$
UNTIL X%>100
PRINT TIME
END
.workitout x$,y$
X%=X%+1
=x$+" you "+y$+" with the hat"
```

```
254
```

The next example puts the strings into variables and passes only the memory LOCATION of each string to the function sub-procedure. This method is faster as indicated by the time print out.

Example

```
.START
X%=0
TIME=0
x$="Hello"
y$="there"
REPEAT
   X$=FNworkitout (x$,y$)
   PRINT X$
UNTIL X%>100
PRINT TIME
END
.workitout^x$,y$
X%=X%+1
=x$+" you "+y$+" with the hat"
```

```
239
```

Notes

Parameters must be enclosed in brackets, as shown.

When parameters are passed as variables they should be preceded by a circumflex ^ symbol after the label. If this symbol is omitted, the program time might be increased.

The examples, above, are trivial.

In fact it is unnecessary to pass parameters unless the procedure is called by several sections of the program which use different variable names for the parameter(s) passed.

See PROC, GOSUB (also the example in DATA)

FOR ... NEXT

repeat an action

This command defines a loop operation.

Syntax

FOR<numericvar>=<number>TO<number>[STEP<number>]

Example

```
.START
E$=""
FOR X%=65 TO 122 STEP 2
   IF X%>90 AND X%<97 THEN E$=+"*":NEXT
E$=+CHR$(X%)
NEXT
PRINT E$

ACEGIKMOQSUWY***acegikmoqsuwy
```

Notes:

If STEP is omitted then a step of +1 is assumed.

Negative steps may be used.

FORMAT

set format

This function defines the displayed format of numbers and dates.

Syntax
FORMAT [<int1>],[<int2>],[<int3>],[<int4>]

<int1> sets the date format to one of six permutations.

0 - default. Full date printed in words and numbers.

For the other five possible date formats see the example on the following page.

<int2> selects the type of format for numbers.

0 - General format (default)
1 - Exponential format
2 - Fixed decimal point position

<int3> sets the number of digits to be printed.

For General format: sets number of digits (range 1 to 10 but 0 is also interpreted as 10. Default value is 9). If int3 is 1 then neither the decimal point nor any digits following it are printed (and the value is rounded up or down to the nearest integer). This "pointless" format is also used if there would be only zeros after the decimal point.

For Exponential format: sets the number of digits. The last digit is rounded up or down as necessary. Zeros after the decimal point are NOT suppressed.

For fixed decimal point format; sets the number of digits after the decimal point. The last digit is rounded up or down as necessary. Zeros after the decimal point are NOT suppressed (indeed, in the case of an integer, they will be added!).

Range is 0 - 9. A larger int3 will be interpreted as 9.

<int4> sets the length of the string to be printed.

Values are right justified within this string and, if the number of digits is less than the string length, then spaces are inserted at the left. (These spaces will be present if the value is converted to a string by means of STR$, consequently use TRIM$STR$ to avoid this effect.

Default is 10 (including the decimal point). In the case of exponentials the string length includes all characters (i.e. includes E-3 for instance). If the string length specified is less than the number of digits set by int2 and int3 then int4 is taken to be equal to the actual string length.

To reset the format to its original default values type

```
FORMAT 0,0,9,10
```

Unspecified parameters are not altered, so:

```
FORMAT,,,
```

does nothing, and:

```
FORMAT,0,9,10
```

resets the number format but leaves the date format as it was last set.

The examples on the following page should clarify the effect of the FORMAT command.

Program **Result**

<left margin position

```
.START
date@=@"9.10.88"
FORMAT64,,, :PRINT date@              09/10/1988
FORMAT65,,, :PRINT date@              09/10/88
FORMAT66,,, :PRINT date@              09/OCT/1988
FORMAT67,,, :PRINT date@              09/OCT/88
FORMAT1,,,  :PRINT date@              9th October '88
FORMAT0,,,  :PRINT date@              9th October 1988
X=7 :PRINT
FORMAT,0,9,6       :PRINT X                7
FORMAT,0,9,2       :PRINT X            7
FORMAT,0,10,12     :PRINT X/3          2.333333333
FORMAT,0,0,12      :PRINT X/3          2.333333333
FORMAT,0,9,15      :PRINT X/3              2.33333333
FORMAT,0,7,15      :PRINT X/3                2.333333
FORMAT,0,5,13      :PRINT X/3                2.3333
FORMAT,0,9,12      :PRINT X/3           2.33333333
FORMAT,0,9,10      :PRINT X/3         2.33333333
Y%=123456789       :PRINT Y%           123456789
PRINT
FORMAT,1,6,18      :PRINT X/1000              7.00000E-3
FORMAT,1,6,17      :PRINT X/1000             7.00000E-3
FORMAT,1,5,17      :PRINT X/1000              7.0000E-3
FORMAT,1,4,17      :PRINT X/1000               7.000E-3
PRINT Y%                                        1.235E8
PRINT
FORMAT,2,6,9       :PRINT X            7.000000
FORMAT,2,2,9       :PRINT X                7.00
FORMAT,2,2,6       :PRINT X             7.00
FORMAT,2,2,4       :PRINT X           7.00
FORMAT,2,2,1       :PRINT X           7.00
Z%=543             :PRINT Z%          543.00
RETURN
```

FREE

return free memory

This function returns the amount of unused memory available.

Syntax
=FREE

Example

```
PRINT FREE<RETURN>

13021
```

GCOL

set graphics colours

This statement sets the foreground and background colours to be used by subsequent graphics commands and determines how they are affected by existing colours.

Syntax
GCOL<int1>,<int2>

<int1> specifies the logical operation, as follows:

0 Plot the colour specified.
1 OR the specified colour with the existing one.
2 AND the specified colour with the existing one.
3 EOR the specified colour with the existing one.
4 Invert the existing colour.

<int2> specifies the logical colour to be used in future:

Numbers up to and including 127 define the graphics foreground colour. Numbers greater than 127 define the graphics background colour.

In a two colour graphics mode (MODE 0 and 4) the following apply:

foreground	background	colour
0	128	black
1	129	white

In a four colour graphics mode (MODE 1 and 5) the following apply:

foreground	background	colour
0	128	black
1	129	red
2	130	yellow
3	131	white

In a 16 colour mode (MODE 2) the following apply:

foreground	background	colour
0	128	black
1	129	red
2	130	green
3	131	yellow
4	132	blue
5	133	magenta
6	134	cyan
7	135	white
8	136	flashing black/white
9	137	flashing red/cyan
10	138	flashing green/magenta
11	139	flashing yellow/blue
12	140	flashing blue/yellow
13	141	flashing magenta/green
14	142	flashing cyan/red
15	143	flashing white/black

The colours listed are the default or "logical" colours. Each may be swapped with any other of the 16 possibilities by means of the GCOL statement.

GET

get ASCII value of key

This function causes the program to wait until a key is pressed. The ASCII character code is then returned (but not displayed on screen).

Syntax
=GET

Example

```
.START
REPEAT
G%=GET
IF G%>64 AND G%<91 THEN PRINT "upper case "+CHR$(G%)
IF G%>97 AND G%<123 THEN PRINT "lower case "+UPPER$CHR$(G%)
UNTIL G%=32 OR G%=13
PRINT"END"
```

Example

```
.START
W$=""
REPEAT
  G%=GET
  IF G%>64 AND G%<91 OR G%>96 AND G%<123
            PRINT CHR$G%;
            W$=+CHR$G%
  ENDIF
UNTIL G%=13
```

Notes:
Since all keys pressed are stored in a buffer, it is possible that GET will find a character already in the buffer. For this reason it is usual to clear the buffer with *FX21,0 before GET is used.

See ASC, CHR$, GET$, INKEY, INKEY$, INPUT, INPUT$, INPUTLINE

GET$

get character of key pressed

This function causes the program to wait until a key is pressed. The ASCII character string is then returned (but not displayed on screen).

Syntax
=GET$

Example
```
.START
W$=""
REPEAT
  G$=GET$
  IF G$ IN "ABCDEFGHIJKLMNOPQRSTabcdefghijklmnopqrst"
            PRINT G$;
            W$=+G$
  ENDIF
UNTIL G$="|M"
```

Notes:
Since all keys pressed are stored in a buffer, it is possible that GET$ will find a character already in the buffer. For this reason it is usual to clear the buffer with *FX21,0 before GET$ is used.

See ASC, CHR$, GET, INKEY, INKEY$, INPUT, INPUT$, INPUTLINE

GO END

move file pointer to end

This function moves the file pointer to the START of the LAST record.

GO START

move file pointer to start

This function moves the file pointer to the START of the FIRST record.

The actual record to which the pointer is moved depends upon whether an index is in use and whether USE MARKED is implemented.

Note: When a database is opened the pointer is moved automatically to the START. This is NOT the case if the database is re-opened without having first been closed.

Example

```
.START
READ DB"MYDATAB"VIA"MYINDX"
FIND"picker"
READ REC R()
PRINT"Found ";
PRINT R(2)
GO START
READ REC R()
PRINT"First record is ";
PRINT R(2)
CLOSEALL
END
```

```
Found Pickering,Fred J
First record is Ashton,Jane
```

See END, START, PTR

GOSUB ... RETURN

use subroutine

This command calls a subroutine then returns to the point in the program immediately after the GOSUB command.

Syntax
GOSUB<label>

Example

```
.START
ESCAPE OFF
REPEAT
   CLS
   PRINT''''
   PRINT"MENU"
   PRINT"A TEST 1"
   PRINT"B TEST 2"
   REPEAT
      G$=GET$
   UNTIL G$ IN "ABab" OR ASC(G$)=27
   IF G$ IN "Aa" THEN GOSUB test1
   IF G$ IN "Bb" THEN GOSUB test2
UNTIL ASC(G$)=27
ESCAPE ON
CLS
PRINT"END"
END

.test1
CLS
PRINT'''"This is the first subroutine"
PRINT"Press a key"
G$=GET$
RETURN
```

```
.test2
CLS
PRINT'''"This is the second subroutine"
PRINT"Press a key"
G$=GET$
RETURN
```

Note: Subroutine labels must always be preceded by a full stop.
The word PROC may be used instead of GOSUB and the word ENDPROC may be used instead of RETURN.

See PROC for more details.

GOTO

go to specified label

This statement causes the program to jump to a specific label.

Syntax
GOTO<label>

Example

```
.START
W$=""
REPEAT
   G$=GET$
   IF G$="#"THEN L%=15:GOTO hash
   PRINT G$:
   W$=+G$
   L%=LEN W$
.hash
UNTIL L%>14
```

See GOSUB, PROC

HUNT

search for key

This command searches for a key in the index currently open for reading. If it does not find it then it stops *at the next closest match.* If it reaches the end of file an error will occur. The error can be trapped by using HUNT as a function when it will return FALSE if no close match is found. The key will usually be in lower case.

Syntax
HUNT<string>
HUNT KEY<key array>
HUNT REC<key array>

Example

```
.START
REMOVE R()
DIM R(),14
READ DB"MYDATAB"VIA"MYINDX"
PRINT"Hunt for: ";
A$=LOWER$INPUT$
X%=HUNT A$
IF X%=FALSE
   PRINT"Not found"
   ELSE HUNT A$
   READ REC R()
   PRINT R(2)
ENDIF
CLOSEALL
RETURN
```

HUNT is very similar to FIND and the examples and notes given for FIND apply equally to HUNT. If error trapping is not used then ensure that there is a dummy Record at the end of the database with an appropriate KEY like zzzzzzzz or 9999999.

See FIND

IF ... THEN ... ELSE

perform conditional action

This statement sets a condition and one or more actions to be carried out.

Syntax
IF<condition>THEN<statement>
or

IF<condition>THEN<statement>ELSE<statement>
or

```
IF<condition>
 <statements>
ENDIF
```
or
```
IF<condition
 <statements>
ELSE
 <statements>
ENDIF
```

Example
```
.START
G$=GET$
IF G$ IN "Aa"
   PRINT"A has been pressed"
ELSE
   PRINT"A has not been pressed"
ENDIF
```

Note: If ENDIF is omitted from a lengthy program then the program could continue to run for many lines before the error is noticed and the error message could be very misleading!

IMPORT

copy data into string

This command allows the copying of data from a dormant INTER-package into a string variable.

Syntax
IMPORT<stringvar>,<command>

Example

```
.START
IMPORT A$,"IW.0:GETMARKED"
RETURN
```

This program could be run from any INTER- package (except, in this example, IW.0) to copy a section of marked text from IW.0 into A$.

It is NOT possible to IMPORT from the currently active package.
If, for instance, this short program was in P$ then from IW.1 menu; type :P<RETURN> and the marked text from IW.0 will be copied into A$. Then typing EXPORT A$ would copy it to the cursor position in IW.1 text.

See the chapter "Communicating with INTER-WORD".
See EXPORT

IN

test for string

This operator tests for the presence of a string in another string.

Syntax
<sstring>IN<string>

Example

```
.START
day$="MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY"
PRINT"Type the day"
G$=UPPER$INPUT$
IF G$ IN day$
   PRINT"O.K."
ELSE
   PRINT"Not a day"
ENDIF
```

In the above example, typing the letters "mon" would result in "O.K." since UPPER$ converts then to upper case and "MON" does occur within the string.

See INSTR, ITEM, ITEM$

INDEX$

return index filenames

This function returns a list of index filenames which are enabled for the current WRITE database.

Syntax
=INDEX$

INDEX$ actually puts the list of index names into a string of the form <filename,filename,filename....>. In practice, however, you will know the index names you have used. Consequently this function will be needed only if you develop a database in which the index names can be changed by the user.

Example

```
.START
USE DB"MYDATAB"
A$=INDEX$
C%=COUNTINDEX$
CLOSE"MYDATAB"
PRINT A$
PRINT C%
RETURN
```

results in:

```
MYINDX,INTINDX
            2
```

Note:
Strangely, this function will not work on a database which is opened for READ only. It will, however work on a WRITE only database but, in case this "feature" is altered in later versions of INTER-BASE, it is recommended that you USE the database as in the example.

See DISABLE INDEX, ENABLE INDEX

INFO LEN

return info. block length

This function returns the current length of the information block.

Syntax
=INFO LEN

Example
```
.START
READ DB"MYDATAB"
D%=INFOLEN"MYDATAB"
M%=MAXINFO"MYDATAB"
CLOSE"MYDATAB"
PRINT "Current length ";D%
PRINT "Original length ";M%
PRINT "Space left ";M%-D%
END
```

```
Current length 532
Original length 1024
Space left 492
```

Note that INFOLEN may be written as one word.

See MAX INFO

INITIAL$

make initials upper case

This function converts the first character of each word to upper case and the remainder to lower case.

Syntax
=INITIAL$<sstring>

Example
```
.START
N$="frED floGGiNs"
PRINT INITIAL$N$

Fred Floggins
```

See LOWER$, UPPER$

INKEY

return ASCII code of key

This function returns the ASCII code of a key provided that the key is pressed within a set time limit (specified in centiseconds). If this time limit is exceeded before a key is pressed then the function returns -1.

Syntax
=INKEY<int>

Example
```
.START
REPEAT
   *FX21,0
   PRINT "*";
UNTIL INKEY10<>-1
```

will print **** until a key is pressed.

INKEY allows a program to keep running while looking for a key press whereas GET stops the program until a key is pressed.

INKEY can also be used with a negative parameter to look for a specific key press:

Syntax
=INKEY<-int>

Example
```
.START
REPEAT
   *FX21,0
   PRINT "PRESS <TAB> TO STOP ME!"
UNTIL INKEY-97=TRUE
```

Notes:
Since all keys pressed are stored in a buffer, it is possible that INKEY will find a character already in the buffer. For this reason it is usual to clear the buffer with *FX21,0 before INKEY is used.

See GET, GET$, INKEY$, INPUT, INPUT$, INPUTLINE

INKEY$

return character of key

This function returns the character string of a key provided that the key is pressed within a set time limit (specified in centiseconds). If this time limit is exceeded before a key is pressed then the function returns a null string.

Syntax
=INKEY$(<int>)

Example

```
.START
*FX21,0
REPEAT
   PRINT"*";
   X$=INKEY$(50)
UNTIL X$>""
PRINT X$
RETURN
```

Notes:
Since all keys pressed are stored in a buffer, it is possible that INKEY$ will find a character already in the buffer. For this reason it is usual to clear the buffer with *FX21,0 before INKEY$ is used.

See GET, GET$, INKEY, INPUT, INPUT$, INPUTLINE

INPUT

return string/number

This command allows input of one or more strings or numbers.

Syntax
INPUT{[<prompt string>]<varname>}

Examples

```
INPUTname$
INPUT'name$
INPUT"Type your name "name$"and address "'addr$
INPUT"What is your name";name$
INPUT"Type your name underneath"'name$
INPUTN
INPUT"Please enter a number "N
INPUT"Please enter a number "'N"and your name "name$
```

Notes:
Several prompt strings may be used and several variables input with just one INPUT statement.
The prompt string is optional and may be omitted in which case a question mark will appear.
If a semi-colon is placed after a prompt string a question mark will appear.
If an apostrophe is used the cursor will appear on the next line, without a question mark.
A carriage return terminates the string.
If the string includes a comma, this and any characters following will be lost.
Leading spaces will be stripped.
Numbers will be evaluated on input.

See GET, GET$, INKEY, INKEY$, INPUT$, INPUTLINE

INPUT$

return string

This function returns a single string of keyboard input.

Syntax
=INPUT$

Example
```
.START
PRINT"What is your name ? ";
I$=INPUT$
PRINT I$
```

Notes:
The string is terminated by <RETURN> but does not include a carriage return. Leading spaces are NOT stripped.

See GET, GET$, INKEY, INKEY$, INPUT, INPUTLINE

INPUTLINE

input string/number

This command allows input of strings or numbers.

Example

```
.START
INPUTLINEname$
INPUTLINE"Type your name "name$
INPUTLINE"Type your name";name$
INPUTLINE"Type your name"'name$
INPUTLINEN
INPUTLINE"Type a number "N
INPUTLINE"What is your number";N
INPUTLINE"Type a number beneath"'N
```

Notes:
This command is very similar to INPUT but will accept commas.
The prompt string is optional and may be omitted in which case a question mark will appear.
If a semi-colon is placed after the prompt string a question mark will appear.
If an apostrophe is used the cursor will appear on the next line, without a question mark.
The string is terminated by <RETURN> but does not include a carriage return.

See GET, GET$, INKEY, INKEY$, INPUT, INPUT$

INPUT FROM

read values from file

This command reads values from special files generated by PRINT TO and puts the values into the stated variable(s).

Syntax
INPUT FROM<handle>{,<variable>}

Example

```
.START
OPENOUT"file1"
PRINT TO"file1",65,66,67,68,69
CLOSE"file1"
OPENIN"file1"
REPEAT
   INPUT FROM"file1",x%
   PRINT CHR$(x%);
UNTIL EOF"file1"
CLOSE"file1"
END
```

results in:

```
ABCDE
```

See PRINT TO

INSTALL

install program

This command installs a tokenised program into sideways RAM.

Syntax
INSTALL<RAM No.><stringvar>

See the chapter "ROM Programs".

INSTR

return position of string

Returns the start position of a short string within a longer string.

Syntax
=INSTR(<string>,<sstring>[,<int>])

Example

```
.START
N$="The quick brown fox will fox everyone"
X%=0
REPEAT
   X%=INSTR(N$,"fox",X%+1)
   PRINT X%,
UNTIL X%=0
END
```

```
17        26        0
```

Notes: <int> specifies the start position for the search. If not specified the search begins at the first character. If the short string is not found the function returns zero.

See IN, ITEM, ITEM$

INT

convert to integer

This function converts a decimal number to an integer. The result is never greater than the original value.

Syntax
<num-var>=INT<numeric>

Example

```
.START
X%=INT13.4
PRINT X%
X%=INT-13.4
PRINT X%
X%=-INT(ABS-13.4)
PRINT X%
END

13
-14
13
```

See ABS

ITEM

return item number

This function returns the item number of a short string within a longer string.

Syntax
<num-var>=ITEM(<string>,<sstring>[,<sstring>])

Example

```
.START
N$="One,Two,Three, Four,Five, Six,Seven"
X%=ITEM(N$,"Five")
PRINT X%
X%=ITEM(N$,"Four,Five,"," ")
PRINT X%
END

5
2
```

Notes:
The final parameter is the separator which is assumed to be a comma unless specified.
The example program used a comma as a separator (by default) then used a space as the separator.
The short string must be the complete string which lies between the separators (including spaces, except where the space is the separator).
ITEM will return zero if it does not find the short string.

See IN, INSTR, ITEM$

ITEM$

This function returns a short string from a longer string.

Syntax
<svar>=ITEM$(<string>,<int>[,<sstring>])

Example

```
.START
N$="One,Two,Three, Four,Five, Six,Seven"
I$=ITEM$(N$,2)
PRINT I$
I$=ITEM$(N$,2," ")
PRINT I$
I$=ITEM$(N$,-1," ")
PRINT I$
END
```

```
Two
Four,Five,
Six,Seven
```

Notes:
<int> is the item number of the short string. When <int> is positive the count is made from the start of the longer string and, when negative, from the end.
The final parameter is the separator which is assumed to be a comma unless specified.
ITEM$ will return a null string if the short string does not exist in the position specified.

See IN, INSTR, ITEM

KEY$

return key string

This function returns the entire key string of the current record, using information from the current WRITE index and the current READ database. The key string contains the key and the file pointer after it.

Syntax
<string>=KEY$<record>

Example

```
.START
REMOVE R()
DIM R(),14
READ DB"MYDATAB"VIA"MYINDX"
USE UNMARKED
READ REC R()
WRITE INDEX"MYINDX"
A$=KEY$ R()
CLOSEALL
FOR X%=1 TO LENA$
    PRINTSTR$A$[X%],ASC(A$[X%])
NEXT
```

```
A                        65
s                       115
h                       104
t                       116
o                       111
n                       110
,                        44
J                        74
                        128
                          0
                          7
                         28
```

Notes:
Since the key string contains file pointer information as non-ASCII codes it is inadvisable to print them directly. In the example, for instance, the code 7 causes the computer to beep (VDU7) when printed. Some codes have more disastrous effects!

One way around this problem is to add to the program:

```
key$=A$[1,-4]
ptr$=A$[-4,-1]
```

which will separate the key from the file pointer.

The example index was constructed on only one field. If it had used more than one, then KEY$ would have returned the key for each field, plus the pointer.

See notes about index KEYs under CREATE INDEX, CREATE USER INDEX, CRITERIA, FIND, READ KEY and SORT.

See CREATE INDEX, CRITERIA, FIND, READ KEY, SORT, USE, WRITE

LEFT$

return left hand string

This function returns a specified number of characters from a string, counting from the beginning (left) of the string. In addition it can be used in reverse to define a sub string at the beginning of the string.

Syntax
<sstring>=LEFT$(<string>,<int>)
LEFT$(<string>,<int>)=<sstring>

Example

```
.START
K$="ABCDEF 123456 GHIJKL 789"
A$=LEFT$(K$,5)
PRINT A$
LEFT$(K$,5)="MN"
PRINT K$
END
```

results in:

```
ABCDE
MNF 123456 GHIJKL 789
```

Another way to achieve this is with square brackets, as follows:

```
.START
K$="ABCDEF 123456 GHIJKL 789"
A$=K$[1,5]
PRINT A$
K$[1,5]="MN"
PRINT K$
END
```

results in:

```
ABCDE
MNF 123456 GHIJKL 789
```

See MID$, RIGHT$

LEN

return string length

This function returns the number of characters in a string. It also returns the number of elements in an array.

Syntax
<num-var>=LEN<string>
<num-var>=LEN<array>

Example

```
.START
K$="ABCDEF 123456 GHIJKL 789"
PRINT LENK$
END

            24
```

Example

```
.START
REMOVE R()
DIM R(),14
FOR X%=1 TO 14
   DIM R(X%),5
NEXT
PRINT LENR()
PRINT LEN R(2)
END

            14
             5
```

See TYPE

LGET$

return string from file

This function returns a string of characters from a file up to (but not including) the terminating character. If this terminator is not specified the function will expect a carriage return.

Example

```
.START
OPENOUT"testfile"
BPUT"testfile","123456789abcdefghijklmnopqrstuvwxyz|MABCDEFGHIJKL
MNOPQRSTUVWXYZ"
CLOSE"testfile"
OPENIN"testfile"
A$=LGET$("testfile")
PRINT A$
PRINT
PTR"testfile"=0
A$=LGET$("testfile","Z")
PRINT A$
CLOSE"testfile"
END
```

results in:

```
123456789abcdefghijklmnopqrstuvwxyz
```

(because |M is terminator)

```
123456789abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXY
```

(because Z is terminator)

See BGET, BGET$, BPUT, EOF, EXT, OPENIN, OPENOUT, OPENUP, PT

LINE$

returns a string line

This function is similar to ITEM$ but the separator is always a carriage return.

Syntax
<stringvar>=LINE$(<string>,<int>)
LINE$(<string>,<int>)=<stringvar>

Example

```
.START
A$="This is line 1 |Mand this is line 2 |M so this is line 3 |M
but this is line 4"
L$=LINE$(A$,2)
PRINT L$
PRINT
LINE$(A$,2)="and this is new"
PRINT A$
END
```

results in:

```
and this is line 2

This is line 1
and this is new
 so this is line 3
 but this is line 4
```

See COUNT, ITEM$, WORD$

LOAD

load file

This command loads a specified file into a string variable.

Syntax
LOAD<stringvar>,<filename>

Example
```
LOAD"prog1",P$
```

See SAVE

LOAD RAM

load rom image

This command will load a rom image of the specified filename into a specified sideways RAM.

Syntax
LOAD RAM<int>,<filename>

Example
```
LOAD RAM 5,"ROMimage"
```

See the chapter "Rom Programs"

LOCAL

localise variable

This command allows you to specify any number of variables as local to a procedure or a function. Variables so specified are then kept separate from variables of the same name which are used elsewhere in the program.

The advantage is that procedures and functions may be written without knowledge of what variables are used in the main program and without fear of corrupting these variables.

Syntax

LOCAL<varname>{,[<varname>]}

Example

```
.START
REM main program
z$="A"
PRINT z$
PROCone
PRINT z$
END

.one
LOCAL z$
REM procedure
z$="B"
PRINT z$
ENDPROC
```

results in:

```
A
B
A
```

Notes
The command can result in a loss of speed if a lot of variables are declared local because at the end of the procedure the program will spend some time removing the LOCAL variables.

The command should begin on the first line after the procedure label. It may also be used again on successive lines if a lot of variables need to be declared local.

The command can be used after .START in a program if the entire program is called as a sub-program of another. Take care, however, not to include the sub-program name in the LOCAL statement, otherwise it will run once then remove itself!

Any type of variable may be declared local, including arrays.

See FN, GOSUB, LVAR, PROC

LN

natural logarithm

This function calculates logarithms to the base 2.71828183

Example
```
PRINT LN 0.6
-0.510825624
```

See ACN, ASN, ATN, COS, DEG, EXP, LOG, RAD, SIN, SQR, TAN, PI

LOCK

lock variable

This command prevents the variable from being changed or read in any way. Any reference to a LOCKed variable will generate an error. LOCKing a variable prevents accidental modification by another program.

Syntax
LOCK<variable>

Example

```
.START
UNLOCK A$
A$="YES"
PRINT A$
LOCK A$
```

Notes
To avoid error messages do not LOCK variables unnecessarily.
You can LOCK a few variables which you need to retain, then use the command RVAR to remove all remaining unlocked variables.
Some variables which begin with the underline character are used by the card index and by the program menu. These may be unlocked and read or altered from within a program but not by typing in immediate mode. Examples of such variables are _M% (the screen mode) and _$ (the current filename).

See MENU, RVAR, UNLOCK

LONG REC

return longest record length

This function returns the length of the longest record in the current WRITE database. It can also be used as a command to set the value since INTER-BASE cannot always keep track of the longest record. The value of LONGREC will never, however, be too small.

The following program will print the value of LONGREC from the database. It will then check the actual length of every record. If the value of LONGREC is wrong it will correct it.

Example

```
.START
WRITE DB"MYDATAB"
LR%=LONGREC
PRINT LR%
CLOSE"MYDATAB"
REMOVE R()
DIM R(),14
USE DB"MYDATAB"VIA"MYINDX"
USE UNMARKED
R%=0
WHILE NOT END
   READ REC R()
   RL%=REC LEN R()
   IF RL%>R% THEN R%=RL%
   SKIP
ENDWHILE
IF R%<LR% THEN LONGREC=R%
CLOSE ALL
END

184
```

See MAX REC LEN, REC LEN

LOWER

convert string to lower case

This command converts all alphabet characters in the given string to lower case. The string may be of any length.

Syntax
LOWER<stringvar>

Example

```
.START
A$=STRING$(255,"M")
A$=+A$
PRINT A$
LOWER A$
PRINT A$
END
```

```
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
```

Note: This command cannot be used as a function.
It operates directly on the contents of the specified string.

See LOWER$, UPPER, UPPER$

LOWER$

copy string as lower case

This function produces a copy of the given string in lower case characters.
The string may not be more than 255 characters long.
The string may be copied into itself, as the example shows.

Syntax
<stringvar>=LOWER$<sstring>

Example

```
.START
A$="THIS string IS MOSTLY capITALs"
A$=LOWER$ A$
PRINT A$
END

this string is mostly capitals
```

```
.START
A$=STRING$(255,"M")
B$=LOWER$ A$
PRINT A$
PRINT B$
END
```

```
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMM
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmm
```

See LOWER, UPPER, UPPER$

LOG

logarithm

This function calculates logarithms to the base 10.

Example

```
PRINT LOG 1.5
0.176091259
```

See ACN, ASN, ATN, COS, DEG, EXP, LN, RAD, SIN, SQR, TAN, PI

LVAR

list variable names

This command lists the names of current variables which are not locked.

Syntax
LVAR[<option>{[,<option>]}]

The command will act upon the type(s) of variable specified according to the option words: INT, REAL, DATE, STRING, ARRAY
If no options are given then the command will act upon ALL types of variable which are not locked. In the first example below, B$ is LOCKed while LVAR is used so B$ is not listed.

Example
```
.START
REMOVE R()
DIM R(),8
A$="":G%=0:date@=@"5/6/89":F=1.2
B$="yes"
LOCK B$
LVAR
UNLOCK B$
END

A$    F    G%    P$    R()    date@
```

Example
```
.START
LVAR STRING,ARRAY
END

A$    B$    R()
```

Note LVAR does not list variable names which begin with the underline character.

See LOCK, REMOVE, PVAR, RVAR,

MARK COUNT

return marked record quantity

This function will return the number of MARKed records in a database.

Syntax
=MARK COUNT

Example
```
.START
READ DB"MYDATAB"
X%=MARKCOUNT"MYDATAB"
CLOSE"MYDATAB"
PRINT X%
END

4
```

Notes:
MARK COUNT may be used on a database which is open for READ, WRITE or both (USE).
The space is unnecessary and the function may be written MARKCOUNT.

See MARKED, MARK REC, REC COUNT, UNMARK REC

MARK REC

mark record

This command marks the current record in the WRITE database.

Syntax
MARK REC

Example

```
.START
WRITE DB"MYDATAB"
MARK REC
CLOSE"MYDATAB"
END
```

MARKing is performed because there is no way to delete a record, other than to OVERWRITE it. If a record is amended and found to be too large to fit back in the same position in the file, it is possible to MARK the original (so it can be ignored in future by USE UNMARKED) and to APPEND the amended version to the end of the file. Relevant indexes must then be updated (SORTed) to point to the new file position of the record.

See APPEND, MARK COUNT, MARKED, SORT, UNMARK REC

MARKED

test for marker

This function returns TRUE (-1) if the current record is marked and FALSE (0) if it is not.

Syntax
=MARKED

Example

```
.START
REMOVE R()
DIM R(),14
READ DB"MYDATAB"
USE ALL
WHILE NOT END
   X%=MARKED
   PRINT X%
   IF X%=-1
      READ REC R()
      PRINT"This record is marked"
      PRINT R(2)
   ENDIF
   SKIP
ENDWHILE
CLOSE"MYDATAB"
END
```

Notes:
The database must be open for READing.

See MARK COUNT, MARK REC, UNMARK REC

MATCH

compare strings

This function compares two strings and returns TRUE (-1) if they are identical or FALSE (0) if they are not.

Syntax
=MATCH(<sstring>,<sstring>)

Example

```
.START
A$="WiLlIaMs"
B$="wIlLiAmS"
IF MATCH(UPPER$A$,UPPER$B$)=TRUE
```

```
    PRINT"O.K."
    ELSE PRINT"Different"
ENDIF
END

O.K.
```

Example

```
.START
REMOVE R()
DIM R(),8
R(1)="HELLO"
R(2)="HELLO"
X%=MATCH(R(1),R(2))
PRINT X%
END

-1
```

Example

```
.START
A$="   WILLIAMS"
B$="WI#LI*"
IF MATCH(TRIM$A$,B$)=TRUE
PRINT"O.K."
ELSE PRINT"Different"
ENDIF
END

O.K.
```

Notes:
The function is case sensitive so it might be necessary to convert all characters to either upper or lower case before performing the match, as shown in the first example. Wildcard characters # (one character) and * (remaining characters) may be used in the SECOND string, as shown in the third example. The function is sensitive to spaces. In the third example the leading spaces are TRIMmed off.

See CODE$, LOWER$, TRIM$, UPPER$

MAX INFO

return info. block space

This function returns the length of the space reserved for the information block in the current READ or WRITE database file.

Syntax
=MAX INFO

Example

```
.START
READ DB"MYDATAB"
X%=MAXINFO"MYDATAB"
CLOSE "MYDATAB"
PRINT X%
END

1024
```

Notes:
The space in MAX INFO can be omitted.

See CREATE DB, INFO LEN

MAX REC LEN

return maximum record length

This function returns the maximum record length which can be saved back to the same position. This is equal to the initial record length plus the buffer length originally set by BUFLEN.

Syntax
=MAX REC LEN<record>

Example

```
.START
REMOVE R()
DIM R(),14
USE DB"MYDATAB"VIA"MYINDX"
USE UNMARKED
R%=0
WHILE NOT END
   READ REC R()
   RL%=REC LEN R()
   MRL%=MAX REC LEN R()
   PRINT RL%,MRL%
   SKIP
ENDWHILE
CLOSE ALL
RETURN

173         193
166         174
148         168
179         199
184         204
```

Notes
MAX REC LEN will never be less than REC LEN and will generally be equal to REC LEN + BUFLEN.

If the amended record you wish to save back to the database exceeds MAX REC LEN you must mark the existing record and APPEND the new one to the end of file. The relevant indexes must then be SORTed on the new record.

See BUFLEN, SORT, REC LEN

MENU

call database program

This "command" is actually the name of the Database program string. If it is typed or used in a program the computer will immediately leave INTER-BASE 0 and enter the DATABASE menu.

Example

```
.START
CLS
PRINT'''"Press a key for database"
G$=GET$
ENTER MENU
```

Notes:
On entry to the DATABASE menu the keyboard buffer is cleared. Consequently, it is not possible to send further commands.
Variables are NOT cleared when the DATABASE menu is entered in this way, however the variables used by the DATABASE can be listed by typing:

```
PVAR <RETURN>
```

In addition, once the database is used, there is no guarantee that program variables will not be corrupted.

If you need to return to INTER-BASE 0 without deliberately removing all variables then use *IB.PMENU <RETURN> (must be in upper case) from the DATABASE. If you use the DATABASE menu option 9) this has the effect of closing all open files and of removing all variables, as if you had typed:

```
*IB.PMENU <RETURN>
CLOSE ALL <RETURN>
RVAR      <RETURN>
```

MID$

return middle of string

This function returns part of a string which starts from the first specified character and is the length of the second. If the second is not specified it will return the part of the string from the first specified character to the end. As a command it will replace part of a string with another.

Syntax
<stringvar>=MID$(<string>,<int>[,<int>])
MID$(<string>,<int>[,<int>])=<sstring>

Example
```
.START
A$="ABCDEFGHIJKLM"
B$=MID$(A$,3,4)
PRINT B$
END

CDEF
```

Example
```
.START
A$="ABCDEFGHIJKLM"
B$=A$[3;4]
PRINT B$
END

CDEF
```

Example
```
.START
A$="ABCDEFGHIJKLM"
MID$(A$,3,4)="123456789"
PRINT A$
END

AB123456789GHIJKLM
```

Example

```
.START
A$="ABCDEFGHIJKLM"
A$[3;4]="123456789"
PRINT A$
END

AB123456789GHIJKLM
```

Notes:

As the examples show, MID$ can be replaced by the square brackets method where the figure after the semi-colon indicates the length and the figure before indicates the first character position.

See ITEM, ITEM$, LEFT$, LINE$, RIGHT$, WORD$

MOD

return division remainder

This function returns the remainder after a division, ignoring any whole number result of division. The result is always an integer.

Example

```
.START
X%=14 MOD 5
PRINT X%
END

4
```

(14/5=2 leaving a remainder of 4)

See DIV

MODE

change screen mode

This command alters the display mode of the screen and permits a choice of graphics, text or both combined with a selection of colours, text size and pixel size as indicated in the list below.

Syntax
MODE<int>

Mode	Graphics	Colours	Text
0	640x256	2	80x32
1	320x256	4	40x32
2	160x256	16	20x32
3	none	2	80x25
4	320x256	2	40x32
5	160x256	4	20x32
6	none	2	40x25
7	Teletext		40x25

Notes:
This command may be used within a procedure or function without problems.
The current screen mode may be determined by means of the following program:

```
.START
%A=135
%X=0
%Y=0
M%=(USR(&FFF4)AND&FF0000)DIV&10000
PRINT M%
END
```

See CLG, Colour, DRAW, GCOL, MOVE, PLOT, VDU

MOVE

move graphics cursor

This command moves the graphics cursor to a specified screen position (without drawing a line).

Syntax
MOVE <Xint>,<Yint>

Example
```
.START
MODE2:VDU5
MOVE 800,800
PRINT"START"
MOVE 800,800
DRAW 100,100
PRINT"FINISH"
END
```

See DRAW, GCOL, MODE, PLOT

NAME$

return filename

This function returns the name of the file whose number is specified.

Syntax
NAME$<number>

Example
```
.START
READ DB"MYDATAB"VIA"MYINDX"
```

```
X%=FILES
FOR N%=1 TO X%
   PRINT NAME$ N%
NEXT
CLOSE ALL
```

This particular example would not be very useful since you know what files are open! The specified number may be negative, in which case different information is returned, as follows:

Nº	Information returned
-1	Last database opened for READ or WRITE
-2	Last READ or VIA index opened. (SKIP will currently operate with this index but if this number returns no index, SKIP operates with database).
-3	Index via file.
-4	Read index.
-5	Write index.
-6	Read database.
-7	Write database.

Example

```
.START
READ DB"MYDATAB"VIA"MYINDX"
USE DB"LETLIST"VIA"LETINDX"
WRITE INDEX"LETINDX"
X%=FILES
FOR N%=-1 TO -7 STEP -1
   PRINT N%;NAME$ N%
NEXT
CLOSE ALL
END
```

```
-1LETLIST
-2LETINDX
-3LETINDX
-4LETINDX
-5LETINDX
-6LETLIST
-7LETLIST
```

See FILES

NOT

logic operator

This logic operator is normally used in a conditional statement to produce a test result.

Syntax
<num-var>NOT(<numeric>)

Example

```
.START
INPUT X%
IF NOT(X%=5) THEN PRINT "NOT 5"
REM the brackets are necessary!
END
```

Example

```
.START
mydata$="apple,pear,banana,grapefruit,orange,turnip"
READ mydata$
WHILE NOT(EOD)
   A$=DATA$
   PRINT "The fruit is "+A$
ENDWHILE
UNREAD
END
```

See AND, EOR, OR

ON ERROR

set error trap

This command allows error trapping during program running.

Syntax
ON ERROR<statements>

Example

```
.START
mydata$="apple,pear,banana,grapefruit,orange,turnip"
X%=1
ON ERROR: ON ERROR OFF: PROCtrap
READ mydata$
REPEAT
   X%=X%+1
   A$=DATA$
   PRINT ;X%,"The fruit is "+A$
UNTIL X%>12
UNREAD
ON ERROR OFF
END
.trap
PRINT"Error trapped. Press a key." : G$=GET$
ENDPROC
```

Notes: An error will occur in the example because the terminating value of X% is greater than the count of data in the string. The error is trapped non-fatally, however, and the program will continue.
Errors will not be trapped if a program is run from outside INTER-BASE
The command will not trap fatal errors such as running out of memory!
ON ERROR and ON ERROR OFF must always be used in pairs in the main body of the program and/or in sub procedures.
After an error the program *always* returns to the ON ERROR statement line, *not* the line where the error happened.

See ERL$, ERM$, ERP$, ERR, ERR$, REPORT, STOP

ON ... GOTO

conditionally jump to label

This statement provides a jump to one of a selection of labels.

Syntax
ON<int>GOTO<label>,<label>, ... ,<label> [ELSE<statements>]

Example
```
.START
CLS
PRINT'''"Type 1 2 or 3"
G$=GET$
X%=VALG$
CLS
ON X% GOTO first,second,third ELSE END
END

.first
PRINT"You typed 1"
.second
PRINT"1 and 1 is 2"
.third
PRINT"1 and 2 is 3"
END
```

Notes:
The ELSE statement is optional but, unless the program has safeguards, an "Out of range" error will occur if it is omitted and the <int> result is greater than the number of labels.
Syntax: There must be a space before ELSE and no comma.

ON ... GOSUB

conditionally call subroutine

This statement provides a subroutine call to one of a selection of labels.

Syntax
ON<int>GOSUB<label>,<label>, ... ,<label> [ELSE<statements>]

Example

```
.START
REPEAT
   CLS: PRINT'''"Type 1 2 or 3"
   G$=GET$
   X%=VALG$
   CLS
   ON X% GOSUB first,second,third ELSE toobig
PRINT'"Press a key"
G$=GET$
UNTIL FALSE
END
.first
PRINT"You typed 1"
RETURN
.second
PRINT"You typed 2"
RETURN
.third
PRINT"You typed 3"
RETURN
.toobig
PRINT"Not 1 2 or 3"
RETURN
```

Notes:
The ELSE statement is optional but, unless the program has safeguards, an "Out of range" error will occur if it is omitted and the <int> result is greater than the number of labels.

OPENIN

open file

This function opens the specified file for READ.

Syntax
OPENIN<filename>
OPENIN<handle>
=OPENIN<filename>
=OPENIN<handle>

Example

```
.START
OPENIN"MYFILE"
PTR"MYFILE"=2
FORY%=1TO8
   x%=BGET"MYFILE"
   PRINT CHR$x%;
NEXT
CLOSE"MYFILE"
PRINT'"Press any key"
K$=GET$
RETURN
```

```
            CDEFGHIJ
```

Notes:

1 See example in OPENOUT where "MYFILE" is created.
2 Each byte in the file is "got" as an integer.
3 Pointer PTR is set to zero when the file is closed. If PTR were not specified in this example it would print ABCDEFGH.
4 If reading from an unknown file, take precautions not to PRINT bytes which may affect the screen or printer!

See BPUT, BGET, BGET$, CHAN, CLOSE, EXT, EOF, LGET$, OPENOUT, OPENUP, PTR.

OPENOUT

open file

This function opens the specified file for WRITE.

Syntax
OPENOUT<filename>
OPENOUT<handle>
=OPENOUT<filename>
=OPENOUT<handle>

Example

```
.START
OPENOUT"MYFILE"
BPUT"MYFILE","ABCDEFGHIJ"
CLOSE"MYFILE"
RETURN
```

NOTES:

1 See example in OPENIN where "MYFILE" is read.
2 Each byte in the file is output as an integer.
3 File pointer PTR is set to zero when the file is closed.

See
BGET, BGET$, BPUT, CHAN, CLOSE, EXT, EOF, LGET$, OPENIN, OPENUP, PTR.

OPENUP

open file

This function opens the specified file for READ and WRITE.

Syntax
OPENUP<filename>
OPENUP<handle>
=OPENUP<filename>
=OPENUP<handle>

Example

```
.START
OPENUP"MYFILE"
BPUT"MYFILE","ABCDEFGHIJ"
PTR"MYFILE"=2
FORY%=1TO8
   x%=BGET"MYFILE"
   PRINT CHR$x%;
NEXT
CLOSE"MYFILE"
PRINT'"Press any key"
K$=GET$
RETURN
```

```
     CDEFGHIJ
```

See
BGET, BGET$, BPUT, CHAN, CLOSE, EXT, EOF, LGET$, OPENIN, OPENOUT, PTR.

OR

OR logical operator

OR performs a bit-wise logical OR where those bits which are "1" in either number remain "1" in the answer.

Example

```
PRINT %1010 OR %0011

         11
```

(because 11 in binary is 1011)

Example

```
PRINT 3 OR 8

         11
```

OR can also be used in program statements

Example

```
A$=GET$
IF A$="Y" OR A$="S" THEN PRINT "hello"

hello
```

Example

```
.START
X%=1
REPEAT
  X%=X%+1
  A$=CHR$X%
UNTIL A$="Y" OR X%>100
PRINT X%

   89
```

See AND, EOR, NOT

OSCLI

call operating system

This command allows a star command to be issued, together with a parameter in the form of a string.

Example

```
.START
file%=148
dir%=file% MOD 46
carry%=file% DIV 46
IF carry%>0
   dir$=STR$dir%
   OSCLI"CDIR"+dir$
   OSCLI"DIR"+dir$
ENDIF
```

The above example assumes that a program keeps a count of the number of files in a directory and, when the count reaches 47, creates a new directory and enters it.

Notes:
Beware of issuing star commands which corrupt memory!

Example

```
.START
OSCLI"COMPACT 30 50"
```

OTHERWISE

See CASE ... ENDCASE

OVERWRITE REC

write record

This command allows you to write a record at a specified pointer position.

Syntax
OVERWRITE RECORD<record>

Notes
This command is almost identical to WRITE REC but does not perform any automatic checks. It is readily possible to write a record which is too long, thereby corrupting any record(s) which follow! Before writing a record, the database pointer REC PTR must be set. This command is used if you have modified a record and made it too long to fit in its original position but do not wish to APPEND it to the end of the file. Provided there is another marked record which is longer you may OVERWRITE this one and avoid wasting space on the disc. You must mark the existing record and, after performing the OVERWRITE, re-SORT any relevant index.

See APPEND, MARK REC, REC LEN, REC PTR

PI

constant

PI has the value 3.14159265

Example
To calculate the circumference of a circle whose radius is 6 mm.

```
PRINT "Circumference is ";2*PI*6;" mm"

Circumference is 37.6991118 mm
```

See ACN, ASN, ATN, COS, DEG, EXP, LN, LOG, RAD, SIN, SQR, TAN

PLOT

perform graphics plot

This command allows the drawing of points, lines and triangles.

Syntax
PLOT <int1>,<int2>,<int3>

<int1> defines the drawing action according to the list, below.
<int2>,<int3> are the respective X and Y coordinates which define the finishing point for the plotting.

int1	Action
0	Move relative to last point.
1	Draw line relative to last point in the current graphics foreground colour.
2	Draw line relative to last point in the logical inverse colour.
3	Draw line relative to last point in the current graphics background colour.
4	Move to a point relative to absolute zero. (See MOVE)
5	Draw a line relative to absolute zero in the current graphics foreground colour. (See DRAW)
6	Draw a line relative to absolute zero in logical inverse colour.
7	Draw a line relative to absolute zero in the current graphics background colour.
8-15	As with 0-7 but with the last point in the line omitted in "inverting actions" such as that produced by GCOL4.
16-23	As with 0-7 but produces a dotted line.
24-31	As with 0-7 but produces a dotted line and the last point is omitted.
64-71	As with 0-7 but only a single point is plotted.
80-87	As with 0-7 but plot and fill the triangle defined by this point and the last two visited.

See COLOUR, DRAW, MOVE, GCOL.

POS

return cursor position

This function returns the horizontal screen character position of the cursor.

Syntax
<int-var>=POS

Example

```
.START
line$="Here is some text to edit"
CLS
PRINT''"HERE I AM:";
X%=POS
Y%=VPOS
PRINT'''"NOW I'M HERE"
TAB X%,Y%
PRINT "AND HERE"
END
```

```
HERE I AM:AND HERE



NOW I'M HERE
```

See VPOS

PRINT

print on screen

This command prints the given expression on the screen.

Syntax
PRINT<expression>

Example

```
.START
Y%=34
Z%=78
REMOVE R()
DIM R(),4
R(1)=@"23/6/89"
R(2)="Hello"
R(3)=1.65
R(4)=26
FOR X%=1 TO 4
PRINT R(X%)
NEXT
PRINTY%,Z%
PRINT"DECIMAL 67 IN HEX IS ";~67
PRINT"BINARY 10000001 in decimal is ";%10000001
PRINT'"FINISHED ";
PRINT"NOW"
END
```

```
23rd June 1989
Hello
      1.65
        26
        34        78
DECIMAL 67 IN HEX IS 43
BINARY 10000001 in decimal is 129
FINISHED NOW
```

The screen character position of the printing may be defined by TAB.

Syntax
PRINTTAB(<int1>[,<int2>])

where <int1> is the X coordinate and <int2> is the optional line number.

Example

```
.START
PRINTTAB(8)"This is indented by 8 spaces"
PRINTTAB(5,6)"and this is indented by 5 on the 6th line down."
```

Notes
The apostrophe (') causes a blank line to be printed.
The semi-colon (;) suppresses the normal numeric justification (see FORMAT).
The comma (,) causes tabulation to the next column.
The tilde (~) causes printing of numeric values in hexadecimal.
The percent symbol (%) causes printing of binary values in decimal.

See FORMAT, TAB, VDU

PRINT TO

send value to file

This command sends a specified value to file in a tokenised format, as follows:

Short strings	00+Length+data
Long strings	Hi Length+Lo length+data
Integer	40+4 bytes
Dates	80+3 bytes
Real numbers	FF+5 bytes

Syntax
PRINT TO<handle>{,<value>}

Example

```
.START
OPENOUT"file1"
PRINT TO "file1",65,@(14,6,89),"FRED"
CLOSE"file1"
OPENIN"file1"
X%=1
REMOVE R()
DIM R(),9
REPEAT
   INPUT FROM"file1",R(X%)
   PRINT R(X%)
   X%=X%+1
UNTIL EOF("file1")
CLOSE"file1"
END
```

results in:

```
        65
14th June 89
FRED
```

See INPUT FROM

PROC

call subroutine

This command calls a subroutine.

Example

```
.START
y$="Mister"
G$="JAMES MASON"
PROCsub G$,y$
PRINT d$
H$="JOHN GAUNT"
PROCsub H$,y$
PRINT d$
END

.sub^x$,^y$
LOCAL z$
z$=INITIAL$x$
d$=y$+" "+z$
ENDPROC
```

results in:

```
Mister James Mason
Mister John Gaunt
```

The somewhat trivial example, above, shows how variables can be passed to a procedure under one variable name and received under a different name. In this way a common procedure can serve more than one section of a program. In the example both G$ and H$ are received as x$ which (because it is a parameter following the label) is LOCAL to the procedure and can, therefore, be used elsewhere with impunity. z$ is also declared LOCAL and may be used elsewhere. The result of the action of the procedure is returned to the main program in d$ which is not local. The circumflex (^) tells the procedure to look for the pointer to the location of the variable parameter in memory. If the circumflex is omitted (replaced by a space) then the entire contents of the variable are copied to the

procedure. The latter method is slower and uses more memory. In the extreme case (for instance where a long string variable parameter is used) the program could run out of memory.
If the value to be passed is not a long string and uses the same variable name in the procedure as in the calling program, then there is no need to pass the value as a parameter: indeed it would be slower to do so. Parameters are usually employed, therefore, only where a procedure is called by more than one section of the main program and deals with variables of differing names.

Example

```
.START
Z$=STRING$(255,"W")
REPEAT
   Z$=+Z$
UNTIL LENZ$>7000
PROCsub Z$
END

.sub^x$
LOWERx$
PRINTx$
RETURN
```

Without the circumflex the example above will produce an "Out of room" error message on a BBC B.

Notes:
The key words PROC and GOSUB are mutually interchangeable, as are ENDPROC and RETURN. According to your previous experience you will probably prefer one more than the other and you may even mix them!
Procedures should be situated AFTER the END statement, otherwise the main program could run on into the procedure and chaos would result.
A variable may be placed after RETURN or ENDPROC so that the procedure may also return the result of a function FN without affecting the operation of the procedure when called by GOSUB or PROC (when such a variable would be ignored).

See FN, GOSUB

PTR

read file pointer

This function reads the current position of the file pointer in the specified file.
As a command it can also be used to set the pointer position.

Syntax
<intvar>=PTR<handle>
PTR<handle>=<intvar>

Example

```
.START
OPENIN"MYFILE"
PTR"MYFILE"=5
X$=BGET$"MYFILE"
CLOSE"MYFILE"
PRINT X$
```

Notes:
PTR does not write to the file nor alter it in any way.
PTR may be used with a file open for READ or for WRITE.
PTR is set to zero when the file is closed but opening a file for a second time without first closing it does not alter the value of PTR.

See BGET, BGET$, BPUT, CLOSE, EXT, LGET$, OPENIN, OPENOUT, OPENUP

PVAR

list variables

This command lists the size and the contents of each of the current variables which are not locked.

Syntax
PVAR[<option>{[,<option>]}]

The command will act upon the type(s) of variable specified according to the option words:

INT,REAL,DATE,STRING,ARRAY

If no options are given then the command will act upon ALL types of variable which are not locked.

Example

```
.START
REMOVE R()
DIM R(),8
R(1)=654
R(2)="today"
A$="HELLO":G%=67:date@=@"5/6/89":F=1.2
PVAR
END
```

```
A$          5 "HELLO"
F           1.2
G%          67
P$          150
R()         8
date@       5th June 1989
```

```
.START
PVAR STRING,ARRAY
END
```

```
A$          5 "HELLO"
R()         8
```

Notes:
The command lists each variable in roughly alphabetical order.
(Actually in order of the ASCII value of the variable name. Lower case variables will, therefore, be listed last.)
The command lists the value of each real number and each integer.
The contents of strings less than 20 characters in length are listed.
The length of each string is listed.
The number of elements in each array is listed.
The contents of arrays are not listed.
LOCKed variables are not listed.
Variable names which begin with the underline character are not listed.

See LOCK, LVAR, REMOVE, RVAR

RAD

radian

This function converts an angle in degrees to radians.

Example

```
PRINT RAD 90

1.57079633
```

See ACN, ASN, ATN, COS, DEG, EXP, LN, LOG, SIN, SQR, TAN, PI

RAM SPACE

return selected ram space

This function will return the number of bytes of sideways RAM which has been selected as workspace by the command SELECT RAM.

Syntax
<intvar>=RAM SPACE

Example 1

```
.START
SELECT RAM 6
X%=RAMSPACE
PRINT X%
CLEAR RAM 6
REM deselects RAM
PRINT RAMSPACE
END

16128
0
```

Note SELECT RAM ALL does not select RAM which has already been selected or which contains a recognisable ROM image, such as IB programs. However, nor does it create an error message if it fails to select. Consequently, if you want to select the RAM regardless of whether or not it is already occupied, it should first be cleared. To do this, use the CLEAR RAM ALL command, followed by SELECT RAM ALL. Finally, to check that some RAM space has been selected, check the value of RAMSPACE and act accordingly. These actions are demonstrated in the second example program below.

Example 2

```
.START
CLEAR RAMALL
SELECT RAMALL
X%=RAMSPACE
```

```
IF X%>0
   READ INDEX"MYINDX"
   RESERVE"MYINDX",13*X%/63
   READ DB"MYDATAB"
   RESERVE"MYDATAB",50*X%/63
   CLOSEALL
ENDIF
```

The example reserves RAM workspace, if available, and shares it between the index and the database. Since each 16K RAM provides 63*256 bytes, it is convenient to apportion the available RAM in 63rds.

See CLEAR RAM, RAM STATUS, RESERVE, SELECT RAM

RAM STATUS
ROM STATUS

return status information

This function returns an integer value which gives information about the current use of a bank of sideways RAM or ROM.

Syntax
<intvar>=RAM STATUS<bank number>
<intvar>=ROM STATUS<bank number>

The integer byte thus returned contains the information in the first four bits, as follows:

Bit 0 - If set then the bank contains INTER-BASE program(s).
Bit 1 - If set then the bank is in use as RFS workspace.
Bit 2 - If set then the bank is Read Only (ROM or write-protected RAM).
Bit 3 - If set then the Operating System is using this bank.

(In binary, bit 0 set=1, bit 1 set=2, bit 2 set=4 and bit 3 set=8).

Example

```
.START
FOR R%=0 TO 15
   rom%=RAM STATUS R%
   IF (rom% AND 4) =4
      F$="ROM "
      ELSE
      F$="RAM "
   ENDIF
   IF (rom% AND 1)=1
      H$=" contains INTER-BASE programs."
      ELSE
      IF (rom% AND 2) =2
         H$=" is SELECTed as RFS workspace."
         ELSE
         IF (rom% AND 8) =8
            H$=" is in use by Operating System."
            ELSE H$=" is not in use."
         ENDIF
      ENDIF
   ENDIF
   PRINT F$;R%;H$
NEXT
END
```

See the chapter "ROM Programs"

READ

specify string to be read

This function designates a string for reading. A full explanation is given under DATA.

An example is given under SOUND.

See DATA, RESTORE, UNREAD

READ DB

open database for reading

This function Opens the specified database for read only.

Syntax
READ DB <filename>

Example

```
.START
REMOVE R()
DIM R(),14
READ DB"MYDATAB"
USE UNMARKED
READ REC R()
PRINT"First record is:"
PRINT R(2)
SKIP
READ REC R()
PRINT"Second record is:"
PRINT R(2)
CLOSE "MYDATAB"
END
```

Notes:
When a database is first opened for READ the record pointer is set to the first record. If it is subsequently reopened without being CLOSEd then the pointer will remain where it was. It would, therefore, be necessary to use GO START to reset the pointer.
In this example, since no index is in use, the order of the records may seem arbitrary. In fact the records are read in the order in which they were first appended (although a record subsequently extended may have been marked and re-appended at the end).

See CLOSE, GO START, GO END, READ DB VIA, USE DB, USE UNMARKED, WRITE DB

READ DB .. VIA ..

open database for reading via index

This function Opens the specified database for read only using the specified index which is also opened for read only.

Syntax
READ DB<filename>VIA<filename>

Example
```
.START
REMOVE R()
DIM R(),14
READ DB"MYDATAB"VIA"MYINDX"
USE UNMARKED
READ REC R()
PRINT"First record is:"
PRINT R(2)
SKIP
READ REC R()
PRINT"Second record is:"
PRINT R(2)
CLOSE"MYDATAB"
CLOSE"MYINDX"
END
```

Notes:
READ DB ... VIA ... opens 2 files which must subsequently be closed.
In this example the records are read in the order dictated by the index which is used.

See CLOSE, FILES, READ DB, READ INDEX, USE UNMARKED

READ FIELD

read record field

This command will read a single field of the current record.

Syntax
READ FIELD<field number>,<variable>

Example
```
.START
READ DB"MYDATAB"VIA"MYINDX"
USE UNMARKED
WHILE NOT END
   READ FIELD 2,name$
   PRINT name$
   SKIP
ENDWHILE
CLOSE ALL
END
```

Notes:
The variable must be appropriate to the field type.
This command is useful in not requiring an array.

See TYPE, WRITE FIELD

READ INDEX

open index for reading

This command opens an index for read only.

Syntax
READ INDEX<filename>

Example

```
.START
REMOVE R()
DIM R(),14
READ DB"MYDATAB"
USE UNMARKED
READ INDEX"MYINDX"
GO START
READ REC R()
PRINT"First alphabetical record is:"
PRINT R(2)
USE INDEX"INTINDX"
GO START
READ REC R()
PRINT"First numeric record is:"
PRINT R(2)
CLOSE "MYDATAB"
CLOSE "MYINDX"
CLOSE "INTINDX"
END
```

Notes:
Where only one index is in use the command READ DB VIA is appropriate.
Where more than one index is to be used it is more sensible to open each index as needed.
The last index to be opened will be the one used.

READ INFO

read information block

This command reads information from the database information block.

Syntax
READ INFO<number>,<stringvar>

Notes:
In every database there is an information block. The block reserves fields which may be used to hold information about the structure of the database. The resident card index makes use of these fields by storing information about a database structure when it is created. You can also read this information and use it to replicate the database.
There is an information field for each database field plus an additional field zero which the card index uses as a recognition code, without which the database cannot be read by the card index program. The code is typically 0,"14,7,Neil" where 0 is the information field number; 14 is the number of fields (1 to 14); 7 is the number of lines in the multiple string field; Neil is essential.
To replicate a card index database you must read each information field and write it to the information block in your own database.

See CREATE DB

READ KEY

read record key

This command reads a record key from an index plus the record pointer.

Syntax
READ KEY<array>
READ KEY<stringvar>

Example

```
.START
READ INDEX "MYINDX"
READ KEY key$
FOR X%=1 TO LENkey$
   IF ASCkey$[X%]<33 OR ASCkey$[X%]>122 THEN key$[X%]="."
NEXT
PRINT key$
CLOSE"MYINDX"
END
```

```
ashton,j....
```

Notes:
The actual database field on which this index is based is the name field. In this example the name is Adams,Paul. Since, however, the key is constructed from only the first 8 characters of the database field and converted to lower case, the actual key in the index is adams,pa plus the pointer. The pointer is a four digit number which gives the location of the record in the database. In this example the FOR ... NEXT loop has been introduced to prevent the pointer number from being printed, since this could produce disastrous effects with some numbers. For instance, if the number 7 was present the computer would respond with a beep (VDU7).

Having determined the actual structure of the key plus pointer we can do a better example:

```
.START
REMOVE k()
DIM k(),2
READ INDEX "MYINDX"
READ KEY k()
PRINT"Key is: ";
PRINT k(1)
PRINT"HEX pointer is:"
PRINT k(2)
CLOSEALL
RETURN
```

```
Key is: ashton,j
HEX pointer is:
                        1820
```

Notes:
A typical index will consist of a key (for example the first 8 letters of a name in lower case) plus the record pointer. The structure of an INDEX file on disc can be visualised as something like the following:

```
adams,pa1543 benson,j1603 collins,1671 davidson1733 good,joh1790
```
...etc.

You might construct another index upon postcodes and it would appear:

```
dl1 8da 1671 lu15 6rf1543 ts18 9sw2363 ...etc.
```

All of the pointer numbers will be there but in a different order. Each pointer specifies the first character of the record in the DATABASE file. Taking Mr Adams as our example, his record in the DATABASE file on disc begins at byte 1543 and looks like:

```
Adams,Paul®15 Whincroft Drive,®Luton,®Bedfordshire.®LU15 6RF
```
(where ® represents carriage return).

The INDEX file on disc, "MYINDX" holds the key "adams,pa1543" and an INDEX file "pcodeINDX" might hold the key "lu15 6rf1543". Since both of these keys refer to the same record in the same database, the pointer number must be the same.

See KEY$, READ INDEX

READ REC

read record into array

This command reads a record from the database which currently is open for READ or for USE. If more than one is open the command will refer to the database which was opened last.

Syntax
READ REC<array>

Example

```
.START
USE DB"MYDATAB"VIA"MYINDX"
USE UNMARKED
READ REC jkh()
CLOSE"MYDATAB"
CLOSE"MYINDX"
PRINT"The fourth record is:"
PRINT jkh()
PRINT"The first field is:"
PRINT jkh(1)
PRINT"The second field is:"
PRINT jkh(2)
PRINT"The third field is:"
PRINT jkh(3)
RETURN
```

```
The fourth record is:
         1Ashton,Jane34 Fellows Lane,
Utterwell,
Brampton,
Barnsley.
BA3 4RY

E037-537-7542037-258-942313/4/79 Jane & Dave12/6/59 Dave17/10/60
Jane
The first field is:
```

```
         1
The second field is:
Ashton,Jane
The third field is:
34 Fellows Lane,
Utterwell,
Brampton,
Barnsley.
BA3 4RY
```

Notes:
The record must always be read into an array variable.
It is not strictly necessary to DIM the array beforehand since this will be done automatically according to the types and number of fields in the record.

See READ DB, READ INDEX, READ KEY

REC COUNT

return quantity of records

This function returns the total number of records in the specified database file (or keys in the specified index file), regardless of whether the record is marked.

Syntax
<intvar>=REC COUNT<filename>

Example

```
.START
READ DB"MYDATAB"
PRINT"Total record qty is ";
PRINT REC COUNT"MYDATAB"
CLOSE"MYDATAB"
RETURN

Total record qty is 9
```

Example

```
.START
READ INDEX"MYINDX"
X%=REC COUNT"MYINDX"
PRINT"Total qty of keys is ";X%
CLOSE"MYINDX"
RETURN

Total qty of keys is 5
```

Notes:
Since the index in this example is based upon UNMARKED records, there must be 266-253 MARKED records in the database file.
The relevant database (or index) must be open for READ or WRITE before REC COUNT will work.

See MARK COUNT, READ DB, READ INDEX

REC LEN

return record length

This function returns the length of the current record.

Syntax
<intvar>=REC LEN<record>

Example

```
.START
REMOVE Q()
READ DB"MYDATAB"
READ REC Q()
X%=REC LEN Q()
PRINT"Current record is ";X%;" bytes long"
CLOSE"MYDATAB"
RETURN
```

```
Current record is 166 bytes long
```

The function also works with an index, however this ability is of little use!

Notes:
The function is most useful in determining whether a modified record will still fit in its original position in the disc file. If REC LEN is greater than MAX REC LEN (the available space) then it will not fit. In this case it would be necessary to MARK the existing record in the file and to APPEND the modified record to the end of the disc file. Alternatively, the record could OVERWRITE another existing MARKed record which is longer. In addition any index or indices must be re-sorted to reflect the new position of the record on file (i.e. its file pointer which forms part of its index key will be different).

See APPEND REC, MARK REC, READ REC, REC PTR, SORT REC

REC PTR

return record pointer

This function returns the position in the database file from which the record was read.

Syntax
<intvar>=REC PTR<record>

Example

```
.START
REMOVE Q()
READ DB"MYDATAB"
READ REC Q()
X%=REC PTR Q()
PRINT"Current pointer is ";X%
CLOSE"MYDATAB"
RETURN

Current pointer is 1646
```

Example

```
.START
REMOVE Q()
REMOVE F()
READ DB"MYDATAB"
READ REC Q()
L%=REC LENQ()
PRINT "Record length is ";L%
REPEAT
   USE MARKED
   SKIP
   READ REC F()
   X%=REC PTR F()
   F%=REC LEN F()
UNTIL END OR F%>L%
CLOSE"MYDATAB"
```

```
IF F%>L%
   PRINT"Record length is ";F%
   PRINT"Pointer is ";X%
   PTR"MYDATAB"=X%
   PRINT"OK to overwrite this record?"
   G$=UPPER$GET$
   IF G$="Y" THEN OVERWRITE REC Q()
ENDIF
RETURN
```

```
Record length is 101
Record length is 162
Pointer is 1147
OK to overwrite this record?
```

Notes:
The second example assumes that the first record in the database is to be copied to a new position so that it overwrites an existing MARKED record. The purpose of the program is to determine the length of the first record then to find a marked record which is longer than this.

Normally this operation would be carried out if a record is edited and will not fit back in its original position in the database file on disc.

The only options would then be either to APPEND the record to the end of file or to OVERWRITE an existing MARKED record which is no longer needed.

In either case, any associated index or indices must be re-sorted to reflect the new position of the record in the database file.

Note that the file pointer MUST be set before OVERWRITE is used.

REM

remark

This statement allows remarks to be included in a program.

Syntax
REM<text>

Example

```
.START
REM version 2.10 29/6/89
PRINT"Hello":REM a trivial example
RETURN
```

Notes:
Everything following REM up to the next carriage return is ignored.
The presence of a large number of remarks can reduce the running speed of a program (although subsequent tokenising will remove remarks).
It is not recommended to put REM before START, since it can have disastrous effects when the program is tokenised!

See TOKENISE

REMOVE

remove variable

This command will remove the specified variable.

Syntax
REMOVE<variable>
Example

```
.START
REMOVE Q()
DIM Q(),12
```

Example

```
.START
PRINT TYPE G$
G$="HELLO"
PRINT TYPE G$
G$=""
PRINT TYPE G$
REMOVE G$
PRINT TYPE G$
RETURN

-1
 4
 4
 0
```

Example

```
.START
REMOVEG$,fred$,X%,Y%,R(),date@,jj$,D,Q(),X$
```

Notes:
REMOVE should be used before an array is declared with DIM.
Using REMOVE before redefining long strings will increase program speed.
Using REMOVE to get rid of unwanted variables after use will free extra memory.
The statement LOCAL also has the effect of removing variables when an exit is made from the sub-procedure.
Attempting to REMOVE a variable which does not exist does NOT create an error.
Beware of attempting to REMOVE the current program string!
See DIM, LOCAL, RVAR

REPEAT .. UNTIL

conditionally repeat operation

This conditional statement allows the repetition of statements within a loop until a specified condition is met.

Syntax

```
REPEAT
  <statements>
UNTIL<condition>
```

Example

```
.START
READ DB"MYDATAB"VIA"MYINDX"
USE UNMARKED
REPEAT
   READ REC R()
   SKIP
UNTIL "Simpson" IN R(2) OR END
SKIP-1
X%=REC PTR R()
CLOSE ALL
IF "Simpson" IN R(2)
   PRINT"Found Simpson at ";X%
   ELSE PRINT"Not there"
ENDIF
RETURN

Not there
```

RESERVE

reserve ram workspace

This command acts upon the sideways RAM which has been chosen by SELECT RAM. It can be used to allocate parts of or all of the RAM for use as workspace by one or more specified files. Each file may be a Database file or an Index file. When workspace is reserved in this manner the program will automatically use it as a "RAM disc" by loading part of the file into the RAM. The access time to this part of the file is thereby reduced and alterations to the file can be carried out quickly in RAM then saved back to disc later on. The number of disc read and write operations is reduced and the effect is especially noticeable when it is necessary to scroll through a small number of database "cards". When the end of the information held in RAM is reached, however, this information must be saved back on disc (if it has been altered) and the next RAM full loaded in. At this point there will be a noticeable delay but, overall, the file access time will be reduced.

Syntax
RESERVE<handle>,<size>[,<offset>[,<length>]]

Example

```
.START
SELECT RAM 0
READ INDEX"MYINDX"
RESERVE"MYINDX",16128
```

will reserve one RAM bank for the index "MYINDX"

```
.START
SELECT RAM 5
SELECT RAM 6
READ INDEX"MYINDX"
RESERVE"MYINDX",256*63*2,20000
```

will reserve two RAM banks for the part of the index which begins 20000 bytes from the start of "MYINDX" file on disc.

This method saves time if you know you seldom use the beginning of the index but you would normally omit the ,20000.

```
.START
SELECT RAM ALL
READ INDEX"MYINDX"
RESERVE"MYINDX",4*63*256,0,0
```

The parameter <length> normally defaults to <size> but setting it to zero will load parts of "MYINDX" into RAM only as they are needed.

```
.START
SELECT RAM 0
READ INDEX"MYINDX"
RESERVE"MYINDX",31*256,0,0
READ INDEX"ADDRINDX"
RESERVE"ADDRINDX",32*256,0,0
```

The previous example will reserve roughly half of RAM bank 0 for each index.

Example

```
.START
REMOVE array()
DIM array(),14
READ DB"MYDATAB"VIA"MYINDX"
 SELECT RAM ALL
 RESERVE "MYDATAB",13000
 RESERVE "MYINDX",2000
WHILE NOT END
   READ REC array()
   PRINT array()
   SKIP
ENDWHILE
CLOSE"MYDATAB"
CLOSE"MYINDX"
END
```

Notes:

Each file to which "RESERVE" applies must be opened beforehand.

RAM must be reserved in multiples of 256 bytes. Since the RAM header uses 256 bytes there are 63 lots of 256 available in each bank. If you specify a number which is not a multiple of 256 the program will round it down to the nearest multiple.

The BBC Master has 4 banks of Sideways RAM fitted as standard. In order to use these it is necessary to ensure that certain links inside the machine are set in the correct positions.

The BBC B has no Sideways RAM as standard. However, it is available as an accessory from some suppliers.

Since Sideways RAM can also be used for building and storing Inter-Base program ROMs by using the command INSTALL it is important to avoid clearing such programs unintentionally by CLEAR, SELECT RAM and, especially, SELECT RAM ALL!

See CLEAR RAM, RAM SPACE, RESERVE, SELECT RAM

RESTORE

reset data pointer

This command resets the data pointer to the start of the data string provided that the End Of Data has not been reached.

Syntax
RESTORE

Example

```
.START
list$="hello ,there, John"
READ list$
RESTORE
FOR X%=1 TO 3
   wrd$=DATA$
   PRINT wrd$;
NEXT
RESTORE
READ list$
REPEAT
   wrd$=DATA$
   PRINT wrd$;
UNTIL EOD
UNREAD
RETURN
```

```
hello there Johnhello there John
```

Notes:
It is wise to add dummy data at the end of the string because, unfortunately, RESTORE will not work once the end of data has been reached. It can not, for instance, be used after "UNTIL EOD" otherwise the error message "Out of data" will appear.

Interestingly, the keyword READ acts as RESTORE by itself, so the following example using WHILE NOT(EOD) works!

Example

```
.START
A$="hello,there,John"
FOR X=1 TO 2
READ A$
PRINT
WHILE NOT(EOD)
   PRINT DATA$+" ";
ENDWHILE
NEXT
RETURN

hello there John
hello there John
```

See DATA, READ, EOD, NOT, WHILE ... ENDWHILE

RETURN

return from sub program

This keyword is used to return from a subroutine or from a program.

Syntax
.<label>
<subroutine statements>
RETURN

Syntax
.START
<program statements>
RETURN

This arrangement permits the program to be called as a subroutine from within another program.

Note: The keyword ENDPROC is interchangeable with RETURN.

See END, FN, GOSUB, PROC

RIGHT$

return right hand string

This function returns the specified number of characters from the right hand side of a given string. As a command it can also re-define the right hand side of a string.

Syntax
<stringvar>=RIGHT$(<string>,<int>)
RIGHT$(<string>,<int>)=<string>

Example

```
.START
FRED$="ABCDEFGhijklm"
RIGHT$(FRED$,6)=UPPER$(RIGHT$(FRED$,6))
PRINT FRED$
RETURN

ABCDEFGHIJKLM
```

The square bracket string handling system of IBPL will perform the same task, however:

Example

```
.START
FRED$="ABCDEFGhijklm"
FRED$[-6,-1]=UPPER$FRED$[-6,-1]
PRINT FRED$
RETURN

ABCDEFGHIJKLM
```

See LEFT$, MID$

RND

This function returns a random value from 0 to 1

Syntax
RND=<var>
<var>=RND

Example
```
.START
RND=TIME
X=RND
P.X
RETURN

0.298214438
```

Notes:
RND does not accept any parameters (unlike BASIC).
RND can be seeded, as shown in the example, to produce a pseudo random result dependent upon the value of TIME.

See INT, SGN

ROM STATUS

See RAM STATUS

RUN

execute program

This command will execute the specified program string.

Syntax
RUN<stringvar>

Example
```
RUN prog$ <RETURN>
```

Notes:
There is actually no need to use RUN at all. The example above could be executed simply by typing:

```
prog <RETURN>.
```

See END, ENDPROC, GOSUB, PROC, RETURN

RVAR

remove all variables

This command removes current variables which are not locked.

Syntax
RVAR[<option>{[,<option>]}]

The command will act upon the type(s) of variable specified according to the option words:
INT,REAL,DATE,STRING,ARRAY

If no options are given then the command will act upon ALL types of variable.

Note that RVAR will not remove variables which begin with an underline character. Consequently, if there is a program or a subroutine which you need to retain in memory, you can store it an a variable which begins with an underline character.

Example

```
LOCK P$
RVAR STRING,INT
UNLOCK P$
```

Notes:

RVAR will not remove program strings installed in RAM.

RVAR will empty the default program string (displayed next to menu option 5) but will not remove it.

See LVAR, PVAR, REMOVE

SAVE

save program string

This command saves a string variable as the specified file.

Syntax

SAVE<stringvar>,<filename>

Example

```
SAVE"prog1",P$
```

See LOAD

SAVE RAM

SAVE ROM

save rom image

This command will save a rom image of the given sideways RAM or ROM as the specified filename.

Syntax
SAVE RAM<int>,<filename>
SAVE ROM<int>,<filename>

Example

```
SAVE RAM 5,"ROMimage"
```

See the chapter "Rom Programs"

SELECT RAM

select ram workspace

This command selects one or more banks of sideways RAM as workspace for programs. Up to four banks can be selected. In each bank 256 bytes are occupied by a RAM Filing System (RFS) header code but the remainder is free for use by the program.

Syntax
SELECT RAM<bank number>

SELECT RAM ALL
Up to four banks will be selected if available.

Note:
The error message "RAM already selected" will appear if you have already selected this particular bank. The command
CLEAR RAM<bank number> may be used to deselect it.

See CLEAR RAM, RAM SPACE, RESERVE for more information.

SGN

return sign

This function determines whether a number is positive or negative/zero.

Syntax
<intvar>=SGN<real>

The function returns +1 for a positive number and zero if the number is zero or negative.
Example

```
.START
ddd%=-9
X%=SGNddd%
PRINT X%
RETURN

0
```

Notes:
This function is slightly different from the BASIC equivalent which will return zero only if the number itself is zero and -1 if the number is negative.

See ABS

SHOW

display string

This command displays the specified string but exits immediately without returning a parameter.

Syntax
SHOW<stringvar>,[<int>]

Example

```
.START
A$="THIS IS A LINE OF TEXT FOR THE PURPOSE OF DEMONSTRATING THE
COMMAND SHOW."
SHOW A$
VDU30,10 :REM move cursor to top left +down one line.
PRINT"Press a key"
G$=GET$
PRINT"HELLO"
RETURN
```

Note:
The command is more often used to allow searching for a particular part of a string without allowing editing. Editing can be carried out subsequently, however, as the next example shows:

Example

```
.START
next%=1
A$="THIS IS A LINE OF TEXT FOR THE PURPOSE OF DEMONSTRATING THE
COMMAND SHOW.|M"
A$=+A$:A$=+A$:A$=+A$
find$="TEX"
ESCAPE OFF
REPEAT
   pos%=INSTR(A$,find$,next%)
   IF pos%>0
      SHOW A$,pos%
      x%=POS:y%=VPOS
      VDU30,10 :REM move cursor to top left +down one line.
      PRINT"Edit Y/N?"
      VDU31,x%,y% :REM cursor to x%,y%
      Y$=UPPER$GET$
      IF Y$="Y" THEN EDITA$,(pos%)
      next%=pos%+1
   ENDIF
UNTIL pos%=0 OR ASCY$=27
ESCAPE ON
CLS
VDU30,10
PRINT A$
RETURN
```

See DISPLAY, EDIT, EDITLINE

SIN

sine

This function calculates the sine of an angle in radians.

Examples

```
PRINT SIN 1.5

0.997494987

PRINT SIN RAD 90

         1
```

See ACN, ASN, ATN, COS, DEG, EXP, LN, LOG, RAD, SQR, TAN, PI

SIZE

return quantity of array fields

This function returns the number of fields within an array.

Syntax
<intvar>=SIZE<array>

Example

```
.START
REMOVE R()
DIM R(),10
PRINT LEN R()
PRINT SIZE R()
RETURN

10
 0
```

Example

```
.START
REMOVE R()
DIM R(),10
R(1)="Welcome"
R(2)=34.5
R(3)=9
PRINT LEN R()
PRINT SIZE R()
RETURN

10
 3
```

Note: If the array elements are not used consecutively, an incorrect result will occur

See LEN, TYPE

SKIP

move to next record

This command skips the specified number of records in a file. If no number is specified it defaults to one. The number may be negative.

Syntax
SKIP[<int>]
SKIP[<intvar>]

Example

```
.START
REMOVE R()
DIM R(),14
READ DB"MYDATAB"VIA"MYINDX"
USE UNMARKED
GO START
WHILE NOT END
   READ REC R()
   PRINT R(2)'R(3)'
   SKIP
ENDWHILE
CLOSE ALL
RETURN
```

Example 2

```
.START
X%=-1
REMOVE R()
DIM R(),14
READ DB"MYDATAB"VIA"MYINDX"
USE UNMARKED
GO END
WHILE NOT START
   READ REC R()
   PRINT R(2)'R(3)'
   SKIP X%
```

```
ENDWHILE
CLOSE ALL
RETURN
```

Notes:
A single skip past the start or end of the file will cause both START and END to return TRUE (=-1). Attempting to skip still further will cause an error "Skipped past end".
Note that START and END are interchangeable.
Skip takes account of marked or unmarked records if so instructed.

SKIP may also be used with an index file but, in this case, the instructions USE MARKED etc. have no meaning and will cause an error if used without an open database:

Example

```
.START
REMOVE R()
READ INDEX"MYINDX"
SKIP 4
READ KEY R()
CLOSE"MYINDX"
PRINT R()
RETURN
```

```
pickerin    2212
```

See GO END, GO START, USE ALL, USE MARKED, USE UNMARKED

SORT

sort index file order

This command adds a key to the current index file. The key comprises the first few characters or numbers from the field which is relevant to that index, plus a pointer which is a number defining the position of the actual record in the database file.

Syntax
SORT<sstring>
SORT KEY<array>
SORT REC<array>

As a function it returns FALSE (=-1) if the key is already sorted.
This provides a useful test to avoid errors if you can not be sure that the key has not been sorted.

Syntax
<intvar>=SORT<sstring>
<intvar>=SORT KEY<array>
<intvar>=SORT REC<array>

Notes:
If the first syntax is used (i.e. as a string) then the string must be no longer than the actual key string in the index. By default this is 8 characters long. Also, by convention, the key string should be in lower case unless the index has been constructed otherwise.

Example

```
mystring$="almond"
```

If the second syntax is used (i.e. as a key array) then the key string(s) or number(s) must exist in the array in the same consecutive order that they exist in the index key. With only one index to sort this would require the key to be in the first field of the array.

Example

```
R(1)="almond"
```

If the third syntax is used (i.e. as a record array) then the key string(s) or number(s) must exist in the field(s) in which they normally reside. In this case the number of characters is unimportant as are the contents of the other fields of the array which are not used by the index. In fact these other fields can be left empty.

Example

```
R(2)="almond,kevin"
R(3)="Rubbish"
```

A more comprehensive example can be found under APPEND

Further notes:

SORTing an existing record does not actually move anything in the database disc file or in the index file. It merely alters the file pointer value to reflect the new position of the record in the database file. The file pointer value is part of the record key in the index.

SORTing a new record, however, requires all the keys which are numerically higher in value to be shifted up one position in the index file on disc. (Numerically higher refers to the actual numerical value of a number or to the ASCII values of a string. An index based upon strings should always convert to lower case strings only, otherwise the sort will differentiate between upper and lower case ASCII values, with resultant confusion.)

See APPEND REC, MARK REC, REC LEN

SOUND

produce a sound

This command makes the computer generate sounds.

Syntax
SOUND<int1>,<int2>,<int3>,<int4>

int1 specifies the sound channel, 0,1,2 or 3.
Channel 0 produces only white noise.

int2 specifies the loudness between -15 and 0.
However, the values 1,2,3 or 4 may be used for special effects defined by the relevant envelope which you must define.

int3 specifies the pitch of the note between 0 and 255.
Middle C is 89. A change of 28 will cause the pitch to alter by a perfect fifth. A change of 48 will cause the pitch to alter by an octave.
int4 specifies the duration between 0 and 254 in steps of twentieths of a second. The value of -1 will cause the note to continue until another note is sent to the same channel with flush control set to 1.
int1 is actually more complex than described above and can be programmed as follows:
SOUND <&HSFC>,<int2>,<int3>,<int4>

Each hexadecimal byte has a meaning:

H	0 or 1	Continuation
S	0 to 3	Synchronization
F	0 or 1	Enable Flushing
C	0 to 3	Channel number

Normally H,S and F are zero

If H is 1 then the amplitude and pitch parameters have no effect.
It allows the release segment of a sound to be completed before the next note takes effect.
If S is not zero it permits sound requests on other channels to be queued separately then played simultaneously.

If F is 1 the note will be played immediately, instead of having to join the queue.

Example

```
.START
note$="&0001,-10,-100,20,&0001,-10,-128,20,&0001,-10,-
156,20,&0201,-10,-100,10,&0202,-10,-128,10,&0203,-10,-
156,10,0,0,0,0,0,0,0,0,"
READ note$
FOR X%=1 TO 8
   A%=DATA
   B%=DATA
   C%=DATA
   D%=DATA
   SOUNDA%,B%,C%,D%
NEXT
UNREAD
RETURN
```

See ADVAL, ENVELOPE

SPC

produce spaces

This function provides the specified number of spaces either for printing directly or as a string. The number may be an integer variable.

Syntax
<string>=SPC<int>
<string>=SPC<intvar>

Example

```
.START
F%=9
PRINT SPCF%;"WOW"
PRINT SPC5;"HELLO"
RETURN
```

```
          WOW
      HELLO
```

Example

```
A$="****"+SPC12+"****"
PRINT A$
RETURN

****            ****
```

See STRING$

SQR

square root

This function calculates the square root of a number.

Example

```
PRINT SQR 225

        15
```

See ACN, ASN, ATN, COS, DEG, EXP, LN, LOG, RAD, SIN, TAN, PI

START

define program start point

.START labels the beginning of the program.

See END

STOP

generate error

This command generates a fatal error condition which can not be trapped by an error handling routine.

It is provided to allow exit from a program during development when ESCAPE is disabled. Your program must, however, have some means of allowing entry of the STOP command and of executing it!

See ESCAPE OFF, EXEC, ON ERROR

STR$

convert to string

This function converts a number or a date to a string.

Syntax
<stringvar>=STR$<date>
<stringvar>=STR$<num>

Example

```
.START
date@=@"5/6/89"
date$=STR$date@
D%=VALdate$[-4,-1]
PRINT date$
PRINT STR$D%
RETURN

5th June 1989
1989
```

Example

```
.START
D=3
PRINT D
D$=STR$D
PRINT D$
RETURN

          3
3
```

See ASC, CHR$

STRING$

create string

This function creates a string using the specified short string.

Syntax
<stringvar>=STRING$(<int>,<sstring>)

Example

```
.START
D$=STRING$(22,"*****")
PRINT D$
RETURN

*****************************************************************
*********************************************
```

Note: The string so produced must not exceed 255 characters.

See SPC

SUB@

add value to date

This function adds a specified number of days, months and years to the given date variable and returns the resulting date.

Syntax
SUB@(<date>,<days%>,<months%>,<years%>)

Example
```
.START
date1$="20/6h/89"
date2$=STR$SUB@(@date1$,0,0,38)
PRINT date2$
RETURN

20th June 1951
```

SWAP

swap variable contents

This command swaps the contents of two variables which must be of the same type. It can work on arrays and on single elements of arrays.

Syntax
SWAP<variable>,<variable>

Example
```
.START
A$="HELLO"
B$="GOODBYE"
SWAPA$,B$
```

```
PRINT A$,B$
RETURN
```

```
GOODBYE     HELLO
```

Example

```
.START
REMOVE R()
REMOVE G()
DIM R(),6
DIM G(),6
FOR X%=1 TO 6
   R(X%)=CHR$(64+X%): G(X%)=CHR$(96+X%)
NEXT
PRINT R(): PRINT G()
SWAP R(),G()
PRINT R(): PRINT G()
SWAP R(1),G(6)
PRINT R(): PRINT G()
RETURN
```

```
ABCDEF
abcdef
abcdef
ABCDEF
Fbcdef
ABCDEa
```

TAB

move text cursor

This command moves the text cursor to the specified screen position.

Syntax

TABx,y

Example

```
.START
```

```
CLS
TAB15,12
PRINT"HELLO"
RETURN
```

Note:
TAB may also be used as part of a PRINT statement. In this case the x,y coordinates must be within brackets.

Example
```
.START
CLS
PRINTTAB(15,12)"HELLO"
RETURN
```

See PRINT, POS, VPOS

TAN

tangent

This function calculates the tangent of an angle in radians.

Example
```
PRINT TAN 0.5

0.54630249

PRINT TAN RAD 45

         1
```

See ACN, ASN, ATN, COS, DEG, EXP, LN, LOG, RAD, SIN, SQR, PI

THEN

part of IF...THEN construct

See IF for a full description.

TIME

read/write timer

This function reads the interval timer, returning a value in units of centiseconds.

Syntax
<intvar>=TIME

As a command it can set the interval timer to a specific value.

Syntax
TIME=<int>

Example

```
.START
CLS
PRINTTAB(0,2)"Here is a delay of 5 seconds"
PROCdelay 500
PRINTTAB(0,3)"Here is a delay of 10 seconds"
PROCdelay 1000
PRINT"Finished at time= ";TIME
TIME=0
PROCdelay 1
PRINT'''"Now time = ";TIME
RETURN
```

```
.delay D%
T%=TIME
REPEAT
PRINTTAB(15,10)TIME
UNTIL TIME>T%+D%
ENDPROC
```

Notes:
A delay loop such as this is not very accurate; especially when, as in this example, it is required to perform an additional function (printing TIME on the screen). Consequently, ensure that your conditional loop ends with UNTIL TIME greater than and not equal to!

See TIME$

TIME$

return time and date

This function returns the time and date (BBC Master only).

Syntax
<strvar>=TIME$

As a command it will set the time and date (BBC Master only).

Syntax
TIME$="<day>,<date>.<hours>:<minutes>:<seconds>"
or, separately
TIME$="<day>,<date>"
TIME$="<hours>:<minutes>:<seconds>"

On the BBC B, TIME$ is treated by IBPL as a string of 24 zero characters as if TIME$=STRING$(24,CHR$(0)) and does NOT return an error if used. This fact is useful since a program can be made compatible with both BBC B and Master computers as the following example shows:

Example

```
.START
IF TIME$=STRING$(24,CHR$(0))
   PROCdate
   time$=""
  ELSE
   date$=ITEM$(TIME$,1,".")
   date$=ITEM$(date$,2)
   time$=ITEM$(TIME$,2,".")
ENDIF
date@=@date$
date$=STR$date@
PRINT date$,time$
RETURN

.date
INPUT"Please enter date ";date$
ENDPROC

17th May 1989      01:12:46
```

Example

```
.START
PRINT TIME$
RETURN

Wed,17 May 1989.01:12:46
```

Notes:
You can not differentiate between BBC B and Master by using
IF TYPE TIME$= since the BBC B will return a "Bad name" error.

See ITEM$, TIME

TITLE$

return title

This function returns the title of a database or index file *which must first be opened for READ.*

Syntax
<sstring>=TITLE$<filename>

As a command it defines a title of up to 128 characters in a database or index file *which must be open for WRITE.*

Syntax
TITLE$<filename>=<sstring>

Example

```
.START
WRITE DB"MYDATAB"
TITLE$"MYDATAB"="My Database"
CLOSE"MYDATAB"
READ DB"MYDATAB"
PRINT TITLE$"MYDATAB"
CLOSE"MYDATAB"
RETURN
```

```
My Database
```

Example

```
.START
WRITE INDEX"MYINDX"
TITLE$"MYINDX"="Index of names"
CLOSE"MYINDX"
READ INDEX"MYINDX"
PRINT TITLE$"MYINDX"
CLOSE"MYINDX"
RETURN
```

```
Index of names
```

See READ DB, READ INDEX

TOKENISE

tokenise keywords

This command reduces each keyword in a program to a single byte "token".

Syntax
TOKENISE<program string>

EXAMPLE

```
>TOKENISEmyprog$ <RETURN>
```

Notes:
Tokenising a program string will reduce the amount of memory taken up by the program string and will increase the speed at which it runs.
Tokenising is irrevocable. Once done, the program can not be retrieved or modified. Always save a copy of the program on disc before tokenising!
Tokenising removes all REM statements.
Whilst earlier versions of INTER-BASE allowed REM statements to exist before .START, version 2.0A can not tokenise such a program. Indeed, fatal memory corruption may occur. Ensure that .START is the first line of the program before attempting to tokenise.
The act of tokenising may show up hitherto unnoticed errors.

Example

```
.START
PRINT"HELLO"
RETURN

.demo
PROCtoilet
ENDPROC
```

Tokenising this program results in the error message:

```
Can't find toilet
```

PROCdemo is not actually used, so the fact that PROCtoilet does not exist is irrelevant. The example program will run perfectly well even after tokenising.

However, tokenising may ignore the most obvious errors so ensure that the program is fully debugged before tokenising.

Once a program has been tokenised the lines can not be displayed, consequently error messages are not very meaningful. If a program has an error which you can't find after tokenising, introduce some extra labels around the suspect section then tokenise again. Eventually, by running the program and watching for consecutive labels, you should be able to trace the actual line which is causing the problem. Turning TRACE ON can also be useful.

See the chapter "Rom Programs"

See CLEAR RAM, INSTALL, REMOVE, TRACE

TRACE

print labels/string names

This command causes all labels and string names to be printed, as a program is running, in order to facilitate error finding.

Syntax
TRACE ON
TRACE OFF

TRIM$

trim spaces from string

This function strips spaces from both ends of the given string.

Syntax
<sstring>=TRIM$<sstring>

Example

```
.START
A$="   HELLO   "
PRINT TRIM$A$
RETURN

HELLO
```

TYPE

return variable type

This function returns a number code which represents the type of the given variable.

Syntax
<intvar>=TYPE<variable>

Example

```
.START
PRINT TYPE G$
G$=""
PRINT TYPE G$
REMOVE G$
PRINT TYPE G$
RETURN
```

```
-1
4
0
```

Code	Variable type
-1	Does not exist
0	Has no value (has been REMOVEd)
1	Integer
2	Real number
3	Date
4	String
6	Array
8	Field

Example

```
.START
REMOVE G()
DIM G(),8
G(1)=STRING$(255,"*")
G(1)=+G(1)
PRINT TYPE G(1)
PRINT TYPE G()
RETURN
```

```
4
6
```

See LEN

UNLOCK

unlock variable

This command unlocks a variable, allowing it to be used.
See LOCK

UNMARK REC

remove record marker

See MARK REC

UNREAD

move DATA pointer backward

See DATA, DATA$, READ

UNSORT

remove record key

This command removes a key from the current index.

Syntax
UNSORT<sstring>
UNSORT KEY<array>
UNSORT REC<array>

See APPEND, MARK, SORT

UNTIL

part of REPEAT...UNTIL loop construct

See REPEAT ... UNTIL

UPDATE

update disc files

When handling database files INTER-BASE works in memory and waits until it has a reasonable number of tasks to perform on disc before updating modifications to the records.

Syntax
UPDATE<filename>
UPDATE ALL

Notes:
You can force an update by using CLOSE but this will also reset pointers.
UPDATE permits you to force an update while leaving the files open and the pointers in the same positions.
UPDATE is especially useful if you are using sideways RAM as a disc buffer by means of the command SELECT RAM because, although the RAM file is updated frequently, the disc file is not.
Sensible use of UPDATE can prevent loss of data if the computer is switched off before encountering a CLOSE command.

See CLOSE, SELECT RAM

UPPER

convert string to upper case

This command converts all alphabet characters in the given string to upper case. The string may be of any length.

Syntax
UPPER<stringvar>

Example

```
.START
A$=STRING$(255,"m")
A$=+A$
PRINT A$
UPPER A$
PRINT A$
END
```

```
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmm
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMM
```

Note:
This command cannot be used as a function.
It operates directly on the contents of the specified string.

See LOWER, LOWER$, UPPER$

UPPER$

make upper case copy of string

This function produces a copy of the given string in upper case characters. The string may not be more than 255 characters long.

Syntax
<stringvar>=UPPER$<sstring>

Example

```
.START
A$="THIS string IS MOSTLY capITALs"
A$=UPPER$ A$
PRINT A$
END

THIS STRING IS MOSTLY CAPITALS

.START
A$=STRING$(255,"m")
B$=UPPER$ A$
PRINT A$
PRINT B$
END
```

```
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmm
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMM
```

See LOWER, LOWER$, UPPER

USE ALL

use all records

This command allows both marked and unmarked records in the current database to be accessed and is the default state.

USE MARKED

use marked records

This command allows only MARKED records in the current database to be accessed.

USE UNMARKED

use unmarked records

This command allows only UNMARKED records in the current database to be accessed.

Examples can be found under END, KEY$, READ DB, READ FIELD, READ INDEX, REC PTR, REPEAT, SKIP

USE DB

open database

This command opens the specified database for both READ and WRITE.

Syntax
USE DB<filename>

USE DB ...VIA ...

open database and index

This command opens the specified database for both READ and WRITE and opens the specified index for READ only.

Syntax
USE DB<filename>VIA<index filename>

Example

```
.START
USE DB"MYDATAB"VIA"MYINDX"
WHILE NOT END
   READ REC R()
   R(4)=+"K"
   WRITE REC R()
   SKIP
ENDWHILE
CLOSEALL
RETURN
```

Examples can be found under APPEND, COND, CREATE DB, CREATE INDEX, DISABLE INDEX, ENABLE INDEX, FILES, GO END, INDEX$, LONGREC, MAX REC LEN, NAME$, READ REC, WRITE DB, READ DB

USE INDEX

open index

This command opens the specified index for both READ and WRITE.

Syntax
USE INDEX<index filename>

Example

```
.START
USE DB"MYDATAB"
CREATE INDEX"NEWINDX"ON 2;13
REM index key based on first 13 letters in field 2 of database.
USE INDEX"NEWINDX"
COND$="ITEM$(LOWER$(R(2)),1)=""JONES"""
READ DB"MYDATAB"
WHILE NOT END
            READ REC R()
            IF COND THEN SORT REC R()
            SKIP
ENDWHILE
CLOSEALL
RETURN
```

This example program will create an index based upon all the records beginning with "Jones" in the database.

See APPEND, SORT, USE DB

USR

return register contents

This function returns the contents of the 6502 registers.

The four byte value returned contains the contents of registers P, Y, X and A respectively, with A as the least significant byte.
Example

```
.START
MODE 5
%A=135
M%=USR(&FFF4)
PRINT (M% AND &00FF0000) DIV &10000
RETURN

5
```

This example will print the current screen mode.

See CALL

VAL

return numeric value

This function returns a numeric value corresponding to the specified string.

Syntax
<num>=VAL<sstring>

Example

```
.START
A$="23.4HELLO"
P.VALA$
RETURN

23.4
```

Note:
If the first character in the string is not numeric the result will be zero:

Example

```
.START
B$="H23.4"
P.VALB$
RETURN

0
```

See CHR$, STR$

VDU

send screen command

This statement sends codes to the Visual Display Unit driver.

Syntax
VDU<int>[,<int>,<int>,<int>]

A full description of the VDU statement is beyond the scope of this book. However the following list may prove useful:

```
VDU1,x    Sends the character x to the printer only.
VDU2      Enables the printer.
VDU3      Disables the printer.
VDU4      Causes subsequent text to be written at text
          cursor.
VDU5      Causes subsequent text to be written at graphics
```

	cursor.
VDU6	Disables output to screen.
VDU21	Enables output to screen.
VDU7	Makes a beep sound.
VDU8	Cursor back one character.
VDU9	Cursor forward one character.
VDU10	Cursor down one line.
VDU11	Cursor up one line.
VDU12	Clear text screen.
VDU13	Cursor to left of line.
VDU14	Paged mode on. (Press SHIFT to scroll).
VDU15	Paged mode off.
VDU16	Clear graphics screen.
VDU26	Cancel window
VDU28	Define window (see WINDOW)
VDU30	Cursor home (top left).
VDU31x,y	Cursor to position x,y.
VDU127	Backspace and delete.

VDU23;8202;0;0;0;	Cursor off
VDU23;29194;0;0;0;	Cursor on (mode 7 only)
VDU23;26378;0;0;0;	Cursor on (modes 3 and 6 only)
VDU23;26890;0;0;0;	Cursor on (modes 0,1,2,4 and 5 only)

Example

```
.START
MODE 7
VDU23;8202;0;0;0;
REM Cursor off
PRINT"There is no flashing cursor"
PRINT"Press a key"
G=GET
VDU23;29194;0;0;0;
REMCursor on (mode 7 only)
PRINT"Now there is"
RETURN
```

VPOS

return cursor position

This function returns the vertical screen character position of the cursor.

Syntax
<int-var>=VPOS

Example

```
.START
line$="Here is some text to edit"
PRINT"EDIT THIS:";
X%=POS
Y%=VPOS
cur%=1
width%=39
REPEAT
   TABX%,Y%
   cur%=EDITLINE(line$,width%-X%,cur%,ASC"A",ASC"Z")
UNTIL %C=13
PRINT'"Finished"
```

See EDITLINE, POS, TAB

WHILE ... ENDWHILE

conditionally perform action

This statement provides a conditional loop which, unlike REPEAT ... UNTIL need not execute even once if the condition is not met.

Syntax
WHILE<condition>
    <statements>
ENDWHILE

Example

```
.START
READ DB"MYDATAB"VIA"MYINDX"
USE UNMARKED
WHILE NOT END
   READ REC R()
   PRINT R(2)
   PRINT R(3)'
   SKIP
ENDWHILE
CLOSEALL
RETURN
```

WINDOW

define text window

This command defines a text window.

Syntax
WINDOW<int1>,<int2>,<int3>,<int4>

int1 is the X coordinate of the top left corner
int2 is the Y coordinate of the top left corner

int3 is the width
int4 is the depth

Use VDU26 to cancel the window.

Example

```
.START
A$=STRING$(255,"Z")
A$=+STRING$(255,"A")
A$[82]="Edit-this-lot"
MODE7
PRINT' STRING$(40,"*")
FOR X%=39 TO 0 STEP -39
FOR Y%=2 TO 23
   PRINTTAB(X%,Y%);"*";
NEXT
NEXT
PRINTSTRING$(38,"*")
WINDOW6,6,26,13
EDIT A$
CLS
WINDOW3,10,30,2
EDIT A$,80
VDU26
RETURN
```

See EDIT, EDITLINE, VDU

WORD$

return word from string

This function returns the specified item from the given string.

Syntax
<sstring>=WORD$(<string>,<int>)

As a command it replaces the specified item in the given string with another item.

Syntax
WORD$(<string>,<int>)=<sstring>

The separator between items is always a space.

Example

```
.START
A$="first second third fourth fifth sixth Monday Tuesday"
B$=WORD$(A$,8)+" is the "+WORD$(A$,2)+" day."
PRINT B$
RETURN
```

```
Tuesday is the second day.
```

See COUNT, ITEM, ITEM$, LINE$

WRITE DB

open database for writing

This command opens the specified database for WRITE only.

Syntax
WRITE DB<filename>

Examples can be found under BUFLEN, LONG REC, MARK REC

See USE DB, READ DB

WRITE FIELD

write to record field

This command writes to a single field of the current record.

Syntax
WRITE FIELD<field number>,<variable>

Example

```
.START
USE DB"MYDATAB"VIA"MYINDX"
WHILE NOT END
   READ FIELD1,X%
   X%=X%+1
   WRITE FIELD1,X%
   SKIP
ENDWHILE
CLOSE"MYDATAB"
CLOSE"MYINDX"
```

In this example field 1 (the record number in MYDATAB) is incremented by one in each record.

Note:
The new field MUST be exactly the same length as the existing one.
Whilst this will be so for numeric and date fields, string lengths should be checked.
WRITE FIELD is useful in not requiring an array.

See LEN, READ FIELD, USE DB

WRITE INDEX

open index for writing

This command opens the specified index for WRITE only.

Syntax
WRITE INDEX<index filename>

Examples can be found under KEY$, NAME$, TITLE$
See READ INDEX, USE INDEX

WRITE INFO

write information

This command writes information into the information fields which are in the information block associated with every database file.

Syntax
WRITE INFO<field number>,<sstring>

Notes:
Each database file has an information block. Part of this block is reserved for information fields. Normally each information field will be used by the creator of the database to store information about the nature of the corresponding record field (type, size etc.). In addition, there is an information field number zero which may contain special information. For instance, you may record a code which can be recognised by your own database program.

More details and a comprehensive example can be found under CREATE DB.
See INFO LEN, MAX INFO, READ INFO, TITLE$

WRITE REC

write record

This command writes the contents of an array (the "record") to an existing database file at the current file pointer position.

Syntax
WRITE REC<array>

Example

```
.START
USE DB"MYDATAB"VIA"MYINDX"
WHILE NOT END
   READ REC R()
   R(2)=+",Mr"
   WRITE REC R()
   SKIP
ENDWHILE
CLOSEALL
RETURN
```

Note:
WRITE REC will produce an error if an attempt is made to save a record which is larger than the existing record at that position in the database file.

Examples can be found under APPEND, USE DB

See APPEND, MAX REC LEN, READ REC, REC LEN, REC PTR, USE DB, WRITE DB

About INTER-MAIL

INTER-MAIL is supplied as a ROM image "IMrom" on the example disc. It is a utility program which runs from the INTER-WORD menu.

IMPORTANT
It will work only in a BBC Master (or in a BBC B which has been fitted with an Advanced Disc Filing System conversion and with Screen Shadow Ram). It should work in a BBC B+ with ADFS but has not been tested.

INTER-WORD and INTER-BASE must be fitted.

INTER-MAIL gives you access to a database which can hold names, addresses, telephone numbers and other information.
The database is linked to INTER-WORD so you can type correspondence which automatically contains the date and consecutive reference number. Type the first few letters of the surname and the recipient's name and address is put into the letter for you. When you save the letter on disc, the reference number, name, date and an optional comment are also saved so you can find previous correspondence quickly and simply.
There is also a mailshot facility.

Another ROM image "IUrom" on the disc gives you access to a variety of INTER-MAIL utilities including:
Print individual labels; print labels in columns; list birthdays or anniversaries; create new database; undelete records.

To find out more about this free software please load each of the files "IWmail" into INTER-WORD and read or print out the information.

INDEX