HiSoft BASIC Professional

AMIGA Version

MichTron®

HiSoft BASIC Professional

For the Amiga

HiSoft BASIC Professional

Fast Interactive BASIC Compiler

For the AMIGA

From HiSoft

Published by MICHTRON Inc.

576 South Telegraph
Pontiac, Michigan 48053
(313) 334-5700
BBS: (313) 332-5452

HiSoft BASIC Professional

Fast Interactive BASIC Compiler

Reference Manual

Published by MichTron U.S.A.
576 South Telegraph
Pontiac, Michigan 48053
☎ (313) 334-5700
BBS (313) 332-5452

Software by HiSoft

ISBN 0-944500-22-6

YOUR RIGHTS AND OURS: *This copy of HISOFT BASIC PROFESSIONAL is licensed to you. You may sell your copy without notifying us. However, we retain copyright and other property rights in the program code and documentation. We ask that HISOFT BASIC PROFESSIONAL be used either by a single user on one or more computers or on a single computer by one or more users. If you expect several users of HISOFT BASIC PROFESSIONAL on several computers, contact us for quantity diskounts and site-licensing agreements. Also if you intend to rent this program, or place this program on a BBS, contact us for the appropriate license and fee.*

We think this user policy is fair to both you and us; please abide by it. We will not tolerate use or distribution of all or part of HISOFT BASIC PROFESSIONAL or its documentation by any other means.

LIMITED WARRANTY: *In return for your understanding of our legal rights, we guarantee HISOFT BASIC PROFESSIONAL will reliably perform as detailed in this documentation, subject to limitations here described, for a period of thirty days. If HISOFT BASIC PROFESSIONAL fails to perform as specified, we will either correct the flaw(s) within 15 working days of notification or let you return HISOFT BASIC PROFESSIONAL to the retailer for a full refund of your purchase price. If your retailer does not cooperate, return HISOFT BASIC PROFESSIONAL to us. While we can't offer you more cash than we received for the program, we can give you this choice: 1) you may have a cash refund of the wholesale price, or 2) you may have a merchandise credit for the retail price, which you may apply toward buying any of our other software. Naturally, we insist that any copy returned for refund include proof of the date and price of purchase, the original program disk, all packaging and documentation, and be in salable condition.*

If the disk on which HISOFT BASIC PROFESSIONAL is distributed becomes defective within the warranty period, return it to us for a free replacement. After the warranty period, we will replace any defective program disk for $5.00.

We cannot be responsible for any damage to your equipment, reputation, profit-making ability or mental or physical condition caused by the use (or misuse) of our program.

We cannot guarantee that this program will work with hardware or software not generally available when this program was released, or with special or custom modifications of hardware or software, or with versions of accompanying or required hardware or software other than those specified in the documentation.

Under no circumstances will we be liable for an amount greater than your purchase price.

Please note: Some states do not allow limitations on how long an implied or express warranty lasts, or the exclusion or limitation of incidental or consequential damages, so some of the above limitations or exclusions may not apply to you.

UPGRADES AND REVISIONS: *If you return your information card, we will notify you if upgrades to HISOFT BASIC PROFESSIONAL become available.*

FEEDBACK: *Customer comments are VERY important to us. We think that the use, warranty and upgrade policies outlined above are among the fairest around. Please let us know how you feel about them.*

Many of the program and documentation modifications we make result from customer suggestions. Please tell us how you feel about HISOFT BASIC PROFESSIONAL - your ideas could make the next version better for all of us.

Table of Contents

Appendix A Compiler Options 293

Preface to HiSoft BASIC Professional

Introduction

HiSoft BASIC Professional for the Amiga range of computers is a powerful and modern version of the BASIC language that gives you a totally integrated and interactive environment to make production of your programs easier than it ever has been.

HiSoft BASIC Professional is the result of many years design and programming effort and our goal was to produce a BASIC compiler with the following features:

- interactive edit/compile/run cycle, like an interpreter
- compile AmigaBASIC™ normally without change
- compile Microsoft QuickBASIC™ with minimal change
- compile most flavours of BASIC with little modification
- easily port programs from the PC and the Atari ST
- fast compile time and very fast execution time
- full recursive sub-programs and functions with parameters
- many structured statements like **WHILE...WEND**, **DO...LOOP UNTIL**, **SELECT...CASE** etc.
- full support for the Amiga through the use of libraries
- clear error reporting and correction
- no limits on variable size

Thanks to the power and flexibility of the Amiga and its operating system, we have been able to implement all of these design goals in the **HiSoft BASIC Professional** compiler and the results are explained in detail in this manual.

Please spend some time and effort getting to know and learning how to use the manual so that you can gain the maximum benefit from **HiSoft BASIC Professional**.

The rest of this section explains how to use the manual, whether you are a beginner or an expert, how to use your Amiga to best effect with **HiSoft BASIC Professional** and, finally, we outline the different type styles that we have used throughout the manual to (hopefully) make it easy and enjoyable to use.

How to use this Manual

This manual does not attempt to teach you BASIC, there are hundreds of good books that do this most adequately and we would encourage you to get hold of one (perhaps through your local library) if you are new to BASIC.

Instead, we have designed this manual to tell you about **HiSoft BASIC Professional** on the Amiga. We have packed a great deal of information about the package into the manual and, in order to help you use it efficiently and easily, we will now plot recommended courses through the manual for you, whether you are a beginner to BASIC or a seasoned expert.

A Course for the Beginner

Start by reading the rest of this **Preface**, then **Chapters 1** and **2** in order. You may find that you want to skip some of the explanation in **Chapter 2 (The Tutorial)** but we would encourage you to finish the programs to gain experience in using the editor and compiler.

You should then be ready to begin writing your own programs. You will need to refer to **Chapter 3** for details on how to use the editor and **Chapter 5** for the commands that **HiSoft BASIC Professional** allows you to use.

The Appendices are mainly for reference and you will only need to dip into them occasionally. **Appendix A** details the various options you can use when compiling to get maximum flexibility and/or speed. **Appendix B** will be useful if you have turned off the run-time error messages or if you need more information on the compiler's error messages. You should look at **Appendix C** if converting programs from magazines or from other computers. You will only need **Appendices E** and **F** when you progress to advanced topics while **Appendices D** and **G** are useful for reference at any time.

We hope you find **HiSoft BASIC Professional** easy and friendly to use, please do not hesitate to write to us with any suggestions for improvements and/or alterations.

A Course for Seasoned BASIC Programmers

If you are already familiar with BASIC programming and also with the Amiga computer then we recommend that you use the manual in the following way.

Firstly, finish reading this **Preface** and then **Chapter 1**. If you are eager to start programming then simply double-click on `HiSoft BASIC Professional` (on your working disc!) and refer to **Chapters 3, 4** and **5** for details of how to use the editor and compiler and information on our implementation of BASIC.

However, we would advise you to work through **Chapter 2, The Tutorial**, first since it is through experience that most people learn most efficiently and quickly; it should take roughly 1 hour to complete all parts of the tutorial.

The Appendices are for reference and it is worth glancing through all of them to acquaint yourself with the contents.

Good luck, we hope you find **HiSoft BASIC Professional** a powerful, flexible and easy-to-use development system. Of course, we welcome any written comments you may have on how we might improve both the program and the manual.

System Requirements

- Amiga A500 and upwards with Kickstart 1.2 or higher
- 1 floppy disk drive
- Monochrome or color monitor or TV

It is a general fact of life that the more memory and disk drives you have, the easier your development will be. A500 owners who have not upgraded their RAM may run out of memory when attempting to compile larger programs. To minimise this problem the compiler must be given as much memory as possible in which to run. Normally you should run the system from the Workbench by double-clicking on the HiSoft BASIC Professional icon; using a CLI will eat up more memory and restrict the size of program you can work with on an A500 in interactive mode.

Upgrades to a megabyte of memory are available at very reasonable prices and we strongly recommend this, not just for **HiSoft BASIC Professional** but for general use too.

Typography

In order to make the manual easy to read and to convey the maximum information as clearly as possible, we have adopted certain typefaces and typestyles throughout the manual. These are used as follows:

Typefaces

Bookman	General text
Avant Gard Bold	Chapter and sub-Chapter headings and references to them. Also used to show concept words such as BASIC statements etc.
`Courier`	Used to show something that is typed in at the keyboard or displayed on the screen.

Note that Avant Garde and `Courier` are sometimes used interchangeably in order to aid readability where appropriate.

Typestyles

Bold	Mainly for emphasis.
Italic	Occasionally for emphasis. Mainly used in syntax descriptions to show something to be filled in e.g. `INT (numeric_expression)` where *numeric_expression* is to be replaced by an expression when **INT** is used in a program.

Special Characters

[]	Within syntax descriptions, information enclosed in [] is optional e.g. `[CALL] sub_program_name` means that you can call a sub-program **TEST** by `CALL TEST` or simply by `TEST`.
{}	Within syntax descriptions this indicates a choice of one or more options, each separated by a vertical bar (|) e.g. `PRINT [expression_1] [{;| |,} expression2]...` indicating that expressions in **PRINT** statements may be separated by a semi-colon, a space or a comma.

... Indicates repetition in syntax descriptions.

.
.
.

Vertically-spaced dots show that some part of a program has been omitted

Acknowledgements

The trademarks (both registered and otherwise) of various companies are used throughout this manual. In particular:

Amiga is a trademark of Commodore-Amiga, Inc.

QuickBASIC is a trademark of Microsoft Corp.

TurboBASIC is a trademark of Borland International Inc.

HiSoft BASIC Professional, **Power BASIC**, **Devpac**, **GenAm** and **MonAm** are trademarks of HiSoft.

We acknowledge any other trademark used but not listed above.

We would like to thank the following people for their invaluable help in the production of **HiSoft BASIC Professional** and this manual:

Dave Howorth, Stephan <grin> Somogyi, Sue (for her un-ending patience), Natalie (for keeping us smiling), Julie (for all the hugs, *thanks JC*), the girl with the dog and all the staff at The Old Bell!

Chapter 1
Introduction

Always make a back-up

Before using **HiSoft BASIC Professional** you should make a back-up copy of the distribution disk and put the original away in a safe place. It is not copy-protected to allow easy back-up and to avoid inconvenience. The disk should be backed-up using the Workbench or any back-up utility - before making any backup always write-protect the master to prevent accidental erasure.

Registration Card

Enclosed with this manual is a registration card which you should fill in and return to us after reading the licence agreement. Without it we will not be able to offer you technical support or upgrades.

The README File

As with all HiSoft products **HiSoft BASIC Professional** is continually being improved and the latest details that cannot be included in the manual may be found in the README file This file should be read at this point, by double-clicking on its icon from the Workbench.

Chapter 2
Tutorial

Introduction

This chapter is going to take you through the stages of designing, writing, compiling and running programs in **HiSoft BASIC Professional**.

It will concentrate on how to use the editor and compiler so that you can quickly teach yourself how to use thie fast, interactive programming environment.

We will not try to teach BASIC programming in this chapter, it is solely intended to get you going with **HiSoft BASIC Professional**. There are plenty of good books on programming with AmigaBASIC (remember, **HiSoft BASIC Profesional** is compatible with AmigaBASIC); some of these are listed in the Bibliography – two we would particularly recommend are *AmigaBASIC Inside and Out* by Abacus and *Advanced AmigaBASIC* by Computer Publications. We can supply both of these books.

So, turn on your Amiga, insert your backup of HiSoft BASIC Professional disk 1 and ....

From your Backup of disk 1 double click on the **HiSoft** icon and wait until the editor window appears; a normal Amiga window with a grow box at the bottom right, a close box at the top left and the front-back boxes top right. Now type in this 5-line progam:

```
t=TIMER
DO WHILE TIMER<t+20
     x=MOUSE(1) : y=MOUSE(2)
     IF MOUSE (0) THEN LINE (x,y) - (x+30,y+30),,bf
LOOP
```

It doesn't matter if keywords like TIMER, DO, LOOP etc. are in upper-, lower- or mixed-case; it is good programming style to put them in upper-case but the compiler doesn't care.

This is a very simple little program that reads the current position of the mouse cursor (x=MOUSE(1) : y=MOUSE(2)), waits for you to click on the left mouse button (IF MOUSE(0) ...) and then draws a box at the position of the mouse cursor (LINE (x,y)-(x+30,y+30),,bf). After 20 seconds it times out and finishes (DO WHILE TIMER<t+20 ... LOOP).

Ok, you want to run it? Press AX by which we mean hold the right Amiga key down and press X (lower- or upper-case X). There's another way you can run a program - choose Run from the Program menu. However you have chosen to run the program, you will now see a small window appear with the title Compiling... and, within the window, you will see messages like Tokenising, Parsing and Generating Code - this is the compiler telling you what it is doing. On this small program these messages will appear very quickly.

If you have an error in your program then the compiler will tell you about it by means of an error number and an error message and it will then ask you Continue(Y/N)? We recommend *very strongly* that, while you are getting used to using **HiSoft BASIC Professional**, you always reply N to this question. Replying N will take you back to the editor screen, positioned on the line where the error occurred and with the error message displayed in the bottom right of the window so that you can correct the error and run the program again.

When the program has compiled successfully, you will see some information about the size of the program and then be asked to Press any key. Do this and the program will execute ... move the mouse around the screen, click the left button and see how easy it is to draw little white boxes! After 20 seconds you will be prompted to press a key, then you will be returned to the editor.

Have you noticed how easy it is to write a windowed program? That's because **HiSoft BASIC Professional** automatically opens up a window for your program and closes it down when the program finishes. You can create borderless windows if you want - see the WINDOW statement in the **Command Reference** chapter.

One final point on this simple program - did you notice the DO statement? This is a very flexible structure that is not available in AmigaBASIC but allows you to do many different types of loops with one statement, see the **Command Reference** for details.

Compiling to Disk

Now we'll show you how to create stand-alone programs that you can give-away or sell - but first let's type in a more interesting program. Select New from the Project menu (hold the right mouse button down, move the mouse up to the word Project, down to New and release the right mouse button) - a box will appear saying OK to lose changes?, left-click on the OK button. Type in this, pressing the Return key at the end of each line:

```
Patterns:
CLS
RANDOMIZE TIMER
cx=32 : cy=100 : pi=3.14159265#
xs=cx/4+cx*RND/2 : ys=cy/4+cy*RND/2
COLOR INT(3*RND+1)
FOR k=0 TO 500 STEP 4*pi*RND+1
  x=cx+xs*COS(k) : y=cy+ys*SIN(k)
  IF k=0 THEN PSET(x,y) ELSE LINE -(x,y)
NEXT k
GOTO Pattern
```

Press A X to compile and run your program. If you typed it in exactly as above the compilation should stop with an error: Line number/label missing ... on line 11. Answer N to the Continue(Y/N) message and you will be returned to the editor on line 11. Can you spot the deliberate mistake? That's right, Pattern should be Patterns (plural) because that was the label defined on line 1 of the program. So change Pattern to Patterns by typing an s at the end of the line. Press A X to compile and run the program again - this time it should compile without error, press a key and it will run. Fast, isn't it?

The program repeats itself indefinitely and you may begin to get bored after a while! Move the mouse pointer up to the close box in the top left of the window and left-click - this will take you back to the editor.

Let's save this program to disk - press A S and a file requester will appear (for details see **Chapter 2**). Type in the name of the program Patterns.Bas, say, and then left-click on OK. If the disk is full then you will see a System Request box telling you this; click on Cancel, swap disks and try again.

Ok, you have now saved the source code of your program to disk but you can only run the source code either from the interpreter (you might be interested to try that) or from within the **HiSoft BASIC Professional** environment. If you want to create a stand-alone program that you can run from the Workbench or from a CLI then you must compile your program to disk rather than to memory which is what we have been doing up to now. So type A C which will bring up a large box entitled HiSoft BASIC Professional Compiler Options; you can left-click on the various buttons to turn these options on and off and the options are described fully in the next chapter. For now, simply left-click on the Disk button which is towards the bottom of the box, then left-click on the No box opposite the words Shared Library and then click on Compile.

Your program will now be compiled to disk under the name Patterns - again, if the disk becomes full, find a disk with space and repeat the compilation. When complete, you can go back to the Workbench by typing A Q and double-click on Patterns to run it. Remember to click in the close box to stop the program.

If you type in a new program and then compile it to disk without saving it first then the compiled program will be called NONAME, because, when you compiled it, it had no name.

There you are - you have just created your own, directly run-able program on the Amiga which needs no other files to run. You are free to give it away or sell it now, though you might wait until it's a bit more exciting before trying to sell it to someone!

(You are not free to give away or sell copies of the compiler itself though - that infringes our copyright. You can only re-distribute compiled programs).

If you had not clicked on No after Shared Library then your program would have been compiled in such a form that it would need the hisoftbasic.library to be present in the libs folder of the system and therefore would not be truly stand-alone. The advantage of this is that the compiled programs are much smaller because they are sharing the hisoftbasic.library library, the disadvantage is that they need this other file to be present - you can't win them all!

You are now ready to start typing in programs of your choice, such as programs from magazines or books like those in the Bibliography. However, if you still feel unsure, then we now present two rather longer programs that you may wish to read through and type in to gain more experience. We are going to concentrate on structure and style in these next programs and will be using some concepts (like recursion) that AmigaBASIC does not have. If you are new to BASIC, please read one of the tutorial books listed in the Bibliography before getting too involved in the following two examples.

More Advanced Programming

We are going to develop two programs in order to experience as many of the features of our flavour of BASIC as possible in a short time. The second of these programs is a solution to the famous Towers of Hanoi problem while the first is a simpler example of file handling, building-up and interrogating a small telephone directory.

In both cases, we are going to emphasize the importance of modular, so-called *top-down* programming where you think out the problem very carefully before committing anything to paper, let alone floppy disk!

In the past, this style of programming has not been encouraged by most BASIC interpreters and compilers, the emphasis has been on quick-and-dirty program writing leading to a preponderance of **GOTO**s and **IF...THEN**s making the program resemble a mass of soggy, over-cooked spaghetti! Almost impossible for anybody (including you) to untangle and distinctly un-pallettable.

To be fair to BASIC, the ease of program creation along these lines has done more than anything else to increase the number of people actually programming computers and the ubiquitous home microcomputer would have been far less exciting without an easy-to-use language built in to it.

However, BASICs in the last few years have started to address the spaghetti problem and have added many constructs borrowed from more grown-up languages like Pascal, Modula-2 and C, allowing structured programs to be created which are far easier to read, maintain and which, arguably most importantly, tend to work first time. Our starting point in developing **HiSoft BASIC Professional** was that we should implement the most modern of these BASICs, Microsoft QuickBASIC™, while retaining the ease of interpretative-style development and adding a powerful interface to the Amiga system features.

Thus, **HiSoft BASIC Professional** brings you a modern, extended BASIC on your Amiga, which is also fully compatible with AmigaBASIC itself, and the aim of the rest of this chapter, apart from giving you practice in using **HiSoft BASIC Professional**, is to show you how to develop such modular, structured programs. Of course, you may be an old hand at this already and, if so, please bear with any extended explanation and treat this as an exercise in getting to grips with **HiSoft BASIC Professional** as quickly as possible.

Enough philosophy, let's get down to work.

A Phone Directory

We are going to design and implement a simple telephone directory from which we can reference a particular entry by quoting the area code (STD code to us British people).

We assume that you have already worked through the first part of this chapter and are reasonably familiar with using the editor to create, amend and compile programs.

There will be two programs, one to enter the data and one to interrogate it.

First of all we have to decide on what information we are going to remember and then how we are going to store it on disk.

What's in a Name?

In our telephone directory, it will be useful to remember the first and last names of the person, the country code, area code and telephone number.

The problem with this kind of information is that it is of variable length and therefore, if we simply write the data to our directory as it is, in sequence, we have to search through the entire directory to find a match because we don't know where each record starts (they're variable in length, remember). This will be painfully slow for a large directory (although efficient on space). In this example we shall show a way of getting at a specific entry in a short time although we will use more disk space in doing so.

At this point we need to look at the type of files that **HiSoft BASIC Professional** can handle; there are two specific types: *sequential* and *random-access*.

When writing to or reading from a sequential file, the data is organised in no particular fashion; it is processed in the *sequence* that it was specified relative to the beginning of the file. This means that you cannot write byte number 345 to the file and then bytes 1 through 344; data is written as if it's in a continuous stream.

On the other hand, random-access files consist of *records* that may be accessed in any order. These records have a fixed length which you specify in your BASIC program. A record may also be divided into smaller parts that hold different bits of information.

The Key to Success

Random-access files are precisely what we need for our directory because we can hold a short *key* for each directory entry at the beginning of the file that is associated with the record number of the entry we want. You can then get to each record directly, by giving the key. Normally this key would be worked out from the name by some method (often involving a technique called *hashing*) and may even point to the record number directly without the need for a secondary look-up. However in this case we shall use the Area Code as the key because the programming is simpler, you can always extend it yourself!

Since we have to have a fixed record length, we must decide how a record is to be organised. To go into each record we have the first and last names and the various parts of the telephone number.

Let's set the limit at 10 characters per name, i.e. 10 for the first and 10 for the last. Numeric data isn't quite as easy. The country and area codes are usually relatively short (numeric values smaller than 32767), so they can fit into a (16-bit) short integer each. The actual phone number is 7 digits on the average and must be stored in a (32-bit) long integer. BASIC can't store numeric data in random-access files directly, so we have to convert it into strings first. This will be covered in greater detail later.

Well FIELDed

The statement for defining a record from BASIC is **FIELD**. For our purposes, defining records looks like this:

```
field #1, 10 as firstname$, 10 as lastname$,_
       2 as councode$, 2 as areacode$, 4 as phonum$

field #1, 28 as key$
```

These two statements do the same thing as far as defining the record *size* (28 characters in each case) but allow the record to be referenced either in its entirety through **key$** or in the parts specified in the first **FIELD** statement. The **#1** will be described in detail later.

The string variables that make up a record aren't quite what they seem: they are so-called *fielded* variables. The difference between them and ordinary string variables lies in the way the fielded variables receive their values: the **LSET** and **RSET** statements must be used. **LSET** left-justifies a string and **RSET** right-justifies a string.

These statements must be used to make sure that the string fits into its template; if the string is too long for the fielded variable, it is shortened to fit, if the string is too small, it is moved either to the right or the left of the space allocated for the fielded variable and padded with the necessary amount of spaces.

One final important part of dealing with files of any type is how to actually get at the information in a file. To do this we must **OPEN** a file. This statement specifies the file you wish to access, how you would like to access it and which *channel* should be used. The mode of access depends on the type of file, in our case random-access. The channel number is a feature of BASIC. Once you have opened a file it is not referenced by its name but rather by its channel. The **#1** in the **FIELD** statement above meant that the record is defined for the file at channel 1. You specify the record length in **OPEN** by using **len**. Since you cannot define a record before opening the file, you have to specify the record length here first and then using the **FIELD** statement.

Data Entry Program

With the above preliminaries out of the way we can begin to design the data entry program. The **OPEN** and **FIELD** statements for our program look like this:

```
defint a-z

open "PHONE.DIR" for random as #1 len=28
field #1, 10 as firstname$, 10 as lastname$,_
  2 as councode$, 2 as areacode$, 4 as phonum$
field #1, 28 as key$
```

Type the above into the **HiSoft BASIC Professional** editor (having clicked on **New** on the **Project** menu to clear out any existing program) and save it as `MAKETELE.BAS`.

Now it is time to enter the data for our directory and prepare the key of the file so that we can find a particular entry by its area code. Our, admittedly slightly limited, directory can hold 7 entries. The best place to keep the key information about which entry has which area code is in the first record so it can be accessed quickly. This is the reason the fielded variable **key$** exists. It will hold the information in the format area code, record number, area code, record number etc.

Type the following after the **FIELD** statements:

```
dim allkeys$(8)

for datact=2 to 8
        input "First name: ",frst$
        if len(frst$)=0 then exit for
        lset firstname$=frst$

        input "Last name: ",lst$
        lset lastname$=lst$

        input "Country code: ",cntry
        lset councode$=mki$(cntry)

        input "Area code: ",areac
        lset areacode$=mki$(areac)

        input "Phone number: ",phne&
        lset phonum$=mkl$(phne&)

        allkeys$(datact)=mki$(areac)
        put #1,datact

next datact
```

This is a standard **FOR...NEXT** loop. As you can see, the program asks for the relevant data and uses **LSET** to put the data into the fielded variables. The numeric data is handled slightly differently though.

As mentioned earlier, any data in a record must be a string; numeric data must be converted. To do this we use the **MKI$** and **MKL$** functions. **MKI$** converts a short integer into a 2-byte string whereas **MKL$** converts a long integer into a 4-byte string.

The array **allkeys$()** which needs to be **DIM**med before the **FOR...NEXT** loop keeps a copy of the area code for when we build the key.

The **PUT** statement is used to write an entire record to the file. This is another main difference between random-access and sequential files: records are written to or read in their entirety whereas there is no particular limit to how much data you process at one time when using a sequential file. When the user simply hits [Return] or when 7 entries have been entered, the loop is over.

Now type in the following at the end of the program:

```
        for ctr=2 to 8
                keystr$=keystr$+allkeys$(ctr)+mki$(ctr)
        next ctr

        lset key$=keystr$
        put #1,1

        close #1

end
```

This final part of the program to enter the data generates the key to be used when reading the file. The loop goes through the area codes that are available, adds the record number until all 7 possible entries have been processed. Then the string-variable **keystr$** is put into the fielded **key$** and the first record is written. The file is then **CLOSE**d and the program is over.

Save, compile and run this program and enter some data of your choice so that you can run the interrogation program...

Vee häf vays...

This was only the first part, the data entry. Now we have to be able to interrogate the file to find a directory entry by specifying its area code.

Save the file creation program, **New** the program (on the **Project** menu) and type in the following:

```
        open "PHONE.DIR" for random as #1 len=28

        field #1, 10 as firname$, 10 as lname$,_
         2 as councode$, 2 as citycode$, 4 as phonum$
        field #1, 28 as key$

        get #1,1

        buffer$=key$
```

Save this program as LOOKTELE.BAS.

This code doesn't look too different from the entry program, except for the **GET** statement. This is the counterpart to **PUT**, it reads an entire record, in this case the first record of the directory where our key is located. Also note that we have copied the first record into **buffer$**; this is because we want to keep accessing the keys even though we shall be reading in more records which will corrupt **key$**.

Now append this to the program:

```
input "Which area code would you like ",acode
acode$=mki$(acode)

for srch=1 to 25 step 4
if mid$(buffer$,srch,2)=acode$ then
        rec=cvi(mid$(buffer$,srch+2,2))
        get #1,rec
        print firname$;lname$
        print cvi(councode$); cvi(citycode$); cvl(phonum$)
end if
next srch

        print "No more";acode;"area codes"

        close #1
```

Here we ask for the area code and change it into a string via **MKI$** because the data we get out of the key is also a string.

The search loop then goes through the key record comparing the area code you entered (converted into a string) with the area codes stored in the first record (remember this is now in **buffer$**). These area codes are stored at positions 1 and 2, 5 and 6, 9 and 10 etc. in **buffer$** with the corresponding record numbers stored at positions 3 and 4, 7 and 8, 11 and 12 etc.

If a match is found for the area code you asked for the record number is picked up from **buffer$** and the record is read in.

The record number was stored after the area code, therefore we have to get the next two bytes where the record number of the entry is kept. Then we **GET** the corresponding entry and print it. The **CVI** and **CVL** functions are the counterparts to **MKI$** and **MKL$**. **CVI** converts the value of a two byte string into a short integer; **CVL** converts four-byte strings into long integers.

Finally the entry is printed out and then we loop round to search for more entries associated with this area code until the key record is exhausted.

Save, compile and run the program to check that it works on the data file you created previously.

This was a somewhat artificial and not necessarily useful example of file handling but we hope that it clarified how to program random access files and that it will serve as a basis for extension.

Towers of Hanoi

Firstly, the Towers of Hanoi problem may seem like a rather complicated subject for an example; however, it is a first-class example of designing and writing a structured program and the central solution does not have to be understood in order to appreciate the power and flexibility of this style of programming.

The Problem

Imagine you have 3 poles alongside each other, over which you can place rings of differing sizes. If you have children you may well have such a set of rings and one pole. Initially, you start off with all the rings on the leftmost pole in descending sizes upwards like this:

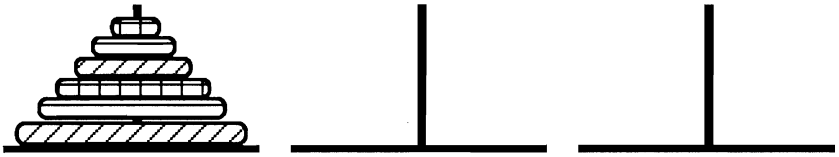


Figure 2.1 Initial Set-up

The object of the puzzle is to move the rings, one at a time, until they are all on the rightmost pole and still in order of descending sizes upwards. The only rule is that you can never place a ring onto one that is smaller.

If you would like to see how this works, double-click on HANOIDEMO which should be on your backup disk; this shows, slowly, how the puzzle is solved for the simple case of 3 rings.

Brain-ache Time

To try and understand how we can solve this problem with a computer let's look at the even simpler case of just two rings:

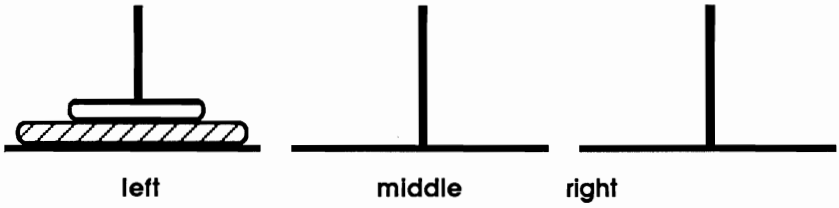


Figure 2.2 Hanoi with 2 rings

We can do it in 3 moves:

1. Move the small ring from the **left** to the **middle**
2. Move the large ring from the **left** to the **right**.
3. Move the small ring from the **middle** to the **right**.

One way of looking at this set of moves is that we have moved the large ring from the left pole (the **source**) to the right pole (the **destination**) using the middle as a **work** pole.

It's also worth noting that the trivial case of 1 ring just involves moving it from **source** to **destination**.

Ok, what about 3 rings?

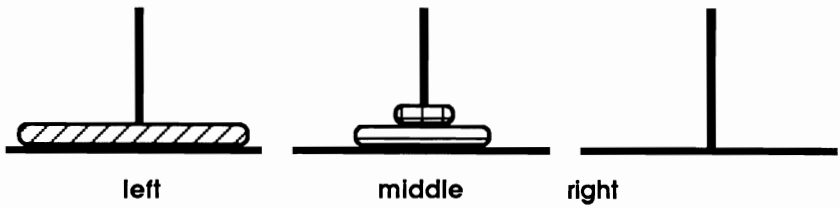


Figure 2.3 Hanoi with 3 rings

This requires 7 moves and you can see how it's done by double-clicking on HANOIDEMO on your backup disk.

However, we can look at it another way by considering the top 2 rings on the left pole as being stuck together to form 1 ring. Then, the solution is just like the case for two rings and involves moving this combined ring to the middle, then the large ring to the right and finally the combined ring to the right. Ok, you say, but that's cheating because you can' t really stick the rings together. No, but it's a way of looking at the problem to try and simplify it; we can move any number of rings by considering them to consist of just 2 rings, the largest ring on the pole and all the others stuck together.

So for 3 rings we consider 1 and 2 stuck together and move them to the middle and this is just like the 2 ring case in **figure 2.2** except that the **source** is the left pole, the **destination** is the middle pole and the **work** pole is the right pole. Then we move the large ring from the left pole to the right pole and finally the two rings on the middle pole (the **source**) are moved to the right pole (the **destination**) via the left pole (the **work** pole).

So, any number of rings always comes down to moving just 2 rings at a time and we have seen how to do this above.

Ok, we're now going to have to take a leap and show you the sub-program that is the heart of the programming solution to the Towers of Hanoi:

```
SUB move(val howmany,val source,val work,val destination)
        IF howmany=1 THEN
                realmove source,destination
        ELSE
                move howmany-1,source,destination,work
                realmove source,destination
                move howmany-1,work,source,destination
        END IF

END SUB
```

What does this do? Let's look at it for 2 rings, you would call this sub-program like this:

```
move 2,left,middle,right
```

i.e. move 2 rings from the **left** to the **right** via the **middle**. **howmany** is therefore 2 to start with and so the **ELSE** clause is actioned; this first says 'call the sub-program **move** again to move 1 ring, swapping the work and destination poles around' i.e. move 1 ring to the middle pole. So the call to **move** at line 5 is **move 1,left,right,middle** and **move** is called within itself! This time **howmany**=1 and so the **ELSE** clause is not actioned but **realmove left,middle** is done instead to actually move a ring from the left pole to the middle pole.

Now we come out of this inner call to **move** and go to line 6 which says **realmove left,right** and actually moves the big ring from the left pole to the right pole. Then line 7 which becomes **move 1,middle,left,right** and again, **move** is called within itself.

This inner call does not do the **ELSE** but does **realmove middle,right** moving the ring on the middle pole on top of the ring on the right pole. The inner call to **move** is exited and then the outer call is exited, we've finished!

Can you see how this would work for 3 rings? Each time **move** is called to move 1 less ring than before until a ring is actually moved by **realmove**; the **move** sub-program is the core to the Towers of Hanoi solution and involves a sub-program that calls itself again and again until something stops the nested calls and goes back up the chain.

A sub-program calling itself is known as *recursion* and is a very clever and powerful programming feature but can give you the most enormous brainache. To keep yourself sane, it's probably best to *believe* that the recursive call is going to work!

If the last few paragraphs have been totally wasted on you, don't despair, it's much easier now we've got **move** out of the way. Enough boring theory; on the next page we'll start typing something in to the computer.

Here we go

Double-click on HiSoft BASIC Professional on your work disk and when the editor is ready type the following, ending each line by typing [Return]:

```
'The Towers of Hanoi program in HiSoft BASIC Professional

DEFINT a-z
CONST left=1,middle=2,right=3

SUB realmove(val source, val destination)
print "Top ring on pole";source;"moved to pole";destination
END SUB

SUB move(val howmany, val source, val work, val
destination)

        IF howmany=1 THEN
                realmove source,destination
        ELSE
                move howmany-1,source,destination,work
                realmove source,destination
                move howmany-1,work,source,destination
        END IF
END SUB

'The actual start

move 3,left,middle,right
```

Now save this to disk under the name MYHANOI.BAS (use **Save** on the **Project** menu) and then **Compile** it from the **Program** menu. Click on **Max Safety** for the sake of caution. If you've made any mistakes typing in the program the compiler will tell you about them and take you back to the editor. You can then correct them and compile again. When the program compiles without error and you're back in the editor, select **Run** from the **Program** menu. You should see a list of 7 moves which corresponds to the solution for 3 rings. Check it out and then hit a key to get back to the editor.

Let's just look at what we've done so far.

defint a-z

all variables are, by default, integers. This ensures speed and compactness.

CONSTants

Values that do not change over the program. We'll be adding to these later.

Sub-program realmove

Just prints out what ring has been moved and where it is moved to. The parameters are **val**ue parameters because we don't want to modify them.

Sub-program move

The core routine (hands up those who understand it yet!). The important things to notice are the use of **val**ue parameters, it would be a disaster if the parameters were modified on each call and the fact that the recursion stops somewhere (when **howmany**=1), otherwise the calls to **move** would go on forever and the routine would disappear up its own ... pole!

Finally, the main program which simply calls **move** to move 3 rings.

See how modular the program is, we've spent a lot of time working out the important sub-program, **move**, and this has paid off; although **move** itself is difficult, the rest of the program is short and clear.

Now let's extend the program so that we can choose the number of rings to move. We need an **INPUT** statement for this but we want to make sure that sensible values are typed in; it would be silly to move just 1 ring and too complicated to move more than about 10 (this upper limit is a bit arbitrary). So, back in the editor, hit ALT T to get to the top of file, click at the end of the **CONST right=3** line and type:

```
[Return]

CONST max_rings=10
```

Remember always to type `[Return]` at the end of a line.

Now hit ALT B to get to the end of the program, cursor-up a few times, click just before the **move** call in the main program and type:

```
DO
        LOCATE 1,2      'at the top of the screen
        INPUT "Number of rings to move: ",num_rings
LOOP UNTIL num_rings>1 AND num_rings<=max_rings
```

Now click after the **3** in **move 3,left,middle,right**, hit [Backspace] and type num_rings.

Save your program and hit Ax to compile and run the program (this is the same as selecting **Run** from the **Program** menu); this compiles and, if successful, runs it automatically after asking you to hit a key. You should be asked Number of rings to move: ?. Firstly answer 1 and then 11 to check that it rejects these answers. Then answer with a sensible number to see if the moves are then made.

Well, that's the Towers of Hanoi solution.

It's not very pretty, though; perhaps we could make look a little more exciting if we added some graphics and showed the rings actually moving on the screen. How on earth are we going to do that?

Adding Graphics

This needs some thought first. The program we have at the moment just tells us which ring to move from pole to pole and we use our intelligence and knowledge of the rules to (mentally) pick up the top ring and move it on top of any rings that are on the destination pole. As we have coded the program, the computer has no idea of the size or the position of the rings and needs to take these into account.

It's a good job we thought about this now rather than later; in fact we should have thought about it before because it's going to involve a major change to **realmove**.

However, we've been foresightful enough to make **realmove** a sub-program which means it is fairly easy to change.

So, we are going to make **realmove** move a ring from one pole to another; to show the ring moving on a trajectory between the poles would be a little difficult so we'll settle for something simpler. We'll define *moving a ring from* ***source*** *to* ***destination*** as meaning:

delete the ring on pole **source**

draw the ring on pole **destination**

We are going to need another sub-program that 'draws' rings at a particular point on the screen of a particular size, either real rings or blank (deleted) rings. **realmove** can then call this to delete a ring and then call it again to draw the moved ring; but how are we going to draw the ring?

The LINE statement comes to our rescue. This allows us to draw a box by using the following syntax:

LINE (x1,y1) - STEP (x2,y2),color,bf

This draws a filled box (because of the bf) with opposite corners (x1,y1) and (x2,y2). The box will be filled with the ink color specified by color so we can use a color of 0 to delete the box.

Right, so we have a sub-program called, say, **drawring** that draws/deletes rings for us. We must tell it on which pole to draw the ring, the width of the ring, what number ring we are drawing (i.e. its position up the pole, number 1 ring will be the bottom ring) and what color to fill the box with.

We're getting on but before we start attacking the keyboard we'd better think a little about what things are going to look like on the screen and how we're going to hold information about the sizes of the rings, where they are etc.

The first thing to do is work out how we are going to store the position and size of the rings so that we can draw them on the screen easily. It seems sensible to hold the information in a two-dimensional array of 3 by **num_rings** (because there are 3 poles and **num_rings** rings to go on them) with each element of the array holding the width of the ring that is on the corresponding pole at the moment. Let's call this array **poles()**.

How wide is the widest ring? Well, the Amiga's screen resolution is 640 wide by 200 high in medium resolution and we have 3 poles across the screen so the widest ring could be 200 pixels wide (we'll call this **max_width**), leaving us 10 pixels between the widest rings (3*200=600, leaving 40 pixels for 4 gaps). If we are having a maximum of **max_rings** rings then the next ring width can be

max_width-max_width\max_rings,

followed by

max_width-2*max_width\max_rings etc.

Ok so far? According to the above, the centre of each pole will start at 110, 320 and 530 pixels across the screen respectively. Call these **pole1**, **pole2** and **pole3**.

What about the height of each ring? The screen is 200 (call this **full_height**) pixels high and (0,0) starts at the top left of the screen and maybe we should leave a gap of 50 pixels top and bottom (call this gap **space**) leaving **full_height-2*space** pixels to draw **max_rings** rings, so each ring can be **(full_height-2*space)\max_rings** (\ means integer division) pixels high. Actually, we'll open a window which will have a title bar which will reduce the effective drawing size but this won't matter much to us; at least we thought about it.

Let's summarize our new identifiers:

poles()	2-dimensional array holding width of rings on each pole
pole1	
pole2	the centre of each pole across the screen, constants
pole3	
space	the gap top and bottom, constant
max_width	the maximum width of a ring, constant
gap	the gap between the widest rings, constant
full_height	the height of the screen in pixels

Let's have a go at writing the **drawring** routine. First of all we'll define the constants, so go to the top of your program (AT), then click after `CONST max_rings=10` and type:

```
[Return]

CONST pole1=110,pole2=320,pole3=530

CONST space=50,max_width=200,gap=10

CONST full_height=200
```

Now type some more at the beginning of the **SUB realmove...** line:

```
'draw or erase a ring
SUB drawring(which_pole,start,size,type)

'x position of start of ring is the pole centre minus
'half the width of the ring

SELECT CASE which_pole
        CASE=1
          xstart=pole1-size\2
        CASE=2
          xstart=pole2-size\2
        CASE=3
          xstart=pole3-size\2
END SELECT

'y position of start of ring
ystart=full_height-space-start*ring_height

'draw ring either filled with white or pattern
IF type=0 THEN
        LINE(xstart, ystart)-STEP(size, ring_height-2),0,bf
ELSE
        LINE(xstart, ystart)-STEP(size, ring_height-2),,bf
END IF

END SUB
```

See how we've used the **SELECT** statement to work out the x co-ordinate of the start of the ring, this is much faster and simpler than using a complicated formula to work it out.

One problem; we've used **ring_height** which is not worked out in this sub-program, it is going to be defined elsewhere (worked out from how many rings there are). Such is the power of **HiSoft BASIC Professional** that **ring_height** will not be available to **drawring** unless we declare it as **SHARED** by the sub-program.

Also, the variables **xstart** and **ystart** will be used only by this sub-program and are therefore local to it and can be declared as **STATIC** (local). To effect both these changes, click before **'x position of start of** ... and enter:

```
SHARED ring_height
STATIC xstart,ystart
[Return]
```

Now let's test out this sub-program.

First, we need to open a window and call **drawring** with some suitable parameters. Go to the bottom of the program (A B), page-up (press [Shift]-↑), click at the end of the **LOOP UNTIL...** line and type:

```
[Return]

ring_height=(full_height-space*2)\max_rings

WINDOW(1),"The Towers of Hanoi in HiSoft BASIC
Professional"

drawring 3,5,100,2
```

The above defines **ring_height** in terms of **max_rings**, opens a window with a title bar with some text in it.

Next we call **drawring** to draw a ring on pole 3, at position 5 (1 is the bottom ring), of width 100 and with a fill pattern. To make sure we don't call **move** while we are testing **drawring**, click in front of the **move num_rings,left,middle,right** line and type a single quote (') to comment out the line.

Let's try it out. Type A X to compile and run, answer any valid number when it asks you Number of rings to move and you should see a ring appear on the screen, if not, check you have typed everything in correctly.

You should now be cautious and save the program before going on.

The next task on the list is to use **drawring** to set up all the rings on pole 1; to do this we must initialise the **poles** array to hold the widths of all the rings on pole 1 and then call **drawring** repeatedly to draw them on the screen.

Click at the beginning of the **drawring 3,5,100,2** line and type [Ctrl]-Y to delete the line and then enter:

```
DIM poles(3,num_rings)
'initialise first pole

FOR i=1 TO num_rings
        poles(1,i)=max_width-(i-1)*(max_width\num_rings)
NEXT i

'draw first pole
FOR i=1 TO num_rings
        drawring 1,i,poles(1,i),2
NEXT i
```

Save the program, compile and run it (AX). Type in how many rings you want and it should draw this many on the first pole.

Nearly there

Well, the last thing to do is to re-write the **realmove** sub-program so that it moves a ring from **source** to **destination** by deleting the source ring and creating one on the destination pole. This should just involve working out the width of the ring to move, where it is on the pole and then calling **drawring** twice, once to erase it and once to re-draw it.

Has a problem occurred to you?

How do we know where the ring is on the pole? When we, as sentient beings(!), did it we just knew to take the *top* ring from the source pole but how does the program know which is the top ring? We're going to have to tell it… which means another array called, say, **highest** with 3 dimensions, one for each pole, that holds the current highest ring for that pole. Rings are numbered from 1 upwards, remember.So, assuming **highest** has been set up, let's type in **realmove**; type AT AF, then PRINT "Top ring [Return] to get to the **realmove** sub-program and enter:

```
[Ctrl]-Y

STATIC ring_width,ystart
'get the width of the ring to move
ring_width=poles(source,highest(source))

'erase source ring
drawring source,highest(source),ring_width,0

'erase ring from array
poles(source,highest(source))=0
DECR highest(source)

'add ring to destination
INCR highest(destination)
poles(destination,highest(destination))=ring_width

'draw destination ring in appropriate style pattern
style highest(destination)
drawring destination,highest(destination),ring_width,2
```

Note how we used **INCR** and **DECR** to add and subtract 1 to and from the highest ring on the destination and source poles respectively.

Ok, let's test it. No...wait, we've forgotten about **highest**; click before the **'initialise first pole** comment and enter

```
DIM highest(3)

highest(1)=num_rings
[Return]
```

Go to the bottom of the program and page-up, click in front of **'move num_rings...** and hit [Del] to get rid of the single-quote and reinstate **move**. Now type ^C to compile and click **Variable Checks Yes** and then press [Return].

Oops... we've got a compiler error (Undefined identifier POLES%), why? Answer N to the Continue? question and we will be placed on the rogue line. You'll see that this is in the **realmove** sub-program and is on the first line in the sub-program that references **poles**. But we declared **poles** as a two-dimensional array in the outer program, why is the compiler complaining?

The answer is that just declaring an array does not make it available to all sub-programs. You either have to declare the array as **DIM SHARED** to make it available to all sub-programs or declare it **SHARED** in the sub-program in which you want to use it. We'd better *share* **highest** as well since **realmove** uses it.

So, move the cursor up until it is in front of the **STATIC ring_width...** line and type:

```
SHARED poles(2),highest(3)
[Return]
```

The 2 in **poles(2)** means this array has 2 dimensions.

Save your program, compile and run it. It should now compile correctly and prompt you for a number of rings. Enter 10 and, lo and behold, it works!

Just when you thought it was safe...

Just to keep you on your toes, we have a couple of optional problems to set you ...

1. The rings move very quickly don't they? Write a Delay sub-program to slow down the movement of the rings; the call to Delay only needs to be added in two places.

2. This is harder. Write the code to show the rings moving from 'pole' to 'pole'; the routine drawring is already there for you - you've just got to work out where to draw the rings moving!

Well, that's the Towers of Hanoi finished. In fact, we've written it so well (and with modesty!) that, with just one change, we can move up to 42 rings (then the stack runs out!). All you have to do is to change the value of the constant **max_rings**. Try it and see... (but don't try anything much higher than 16, it begins to take rather a long time!).

That's the end of the tutorial section. If you are at all confused please go back, read through it again, try out the examples, referring to the **Command Reference** and **Concepts** chapters as often as you like.

We hope you enjoy using **HiSoft BASIC Professional**.

Chapter 3
How to use HiSoft BASIC Professional

Introduction

To enter and compile your programs you need an editor of some sort and a compiler. **HiSoft BASIC Professional** combines both of these functions together in one integrated program, giving an Intuition-driven full-screen editor and a fast, full-specification compiler. It also allows you to run your compiled programs directly from memory without having to quit the program or do a disk access. The fact that all these features are combined in one program means that correcting errors and making changes is as fast as possible without the need for slow disk accesses and other programs.

For users that do not like this type of interactive environment or do not want to use our editor, it is perfectly possible to run the compiler from a CLI (Command Line Interface) - see **Using HiSoft BASIC Professional from the CLI** in **Appendix A**. This chapter details the use of the editor and how to compile programs - it does not detail the language itself, which is covered in the following chapter.

What's on the Disks

HiSoft BASIC Professional is supplied on two disks. Disk 1 contains all you need to start writing and running programs whilst Disk 2 contains sample program source and additional programs.

Disk 1

Disk 1 is based on a standard Workbench 1.3 disk but with the Utilities and Fonts directories removed and these additional files:

HiSoft BASIC Professional	the integrated editor and compiler
HB.Compiler	the compiler itself (required by editor)
HANOIDEMO	example program used in tutorial
hisoftbasic.library	the library file for the compiler (in `libs`)

Disk 2

README	contains latest details not in manual
CALC.BAS	BYTE Calc benchmark
DEMO.BAS	Demonstration program
DUMP.BAS	Hex and ASCII file dumper
HANOI.BAS	Towers of Hanoi
PCWALL.BAS	all the old PCW magazine benchmarks
SIEVE.BAS	BYTE Sieve benchmark

If you wish to use your own Workbench disk configuration, please note that in addition to the *hisoftbasic* library the compiler also needs the *mathieeedoubbas* and *mathieeedoubtrans* libraries, which may be found on disk 1 and copied via the CLI. The *arp* library will also prove useful when using the editor.

To run **HiSoft BASIC Professional**, double click on the `HiSoft BASIC Professional` icon from the Workbench. When it has loaded a menu bar will appear and an empty window will open, ready for you to enter and compile your programs.

The Editor

At the risk of over-simplifying things ... a text editor is a program which allows you to enter and alter lines of text into memory, store them on disk, and load them back again. There are two types of text editors: line editors, which treat each line separately and are often very tricky to use, and screen editors, which display your text a screen at a time. These are normally much easier to use.

The editor section of **HiSoft BASIC Professional** is a screen editor which allows you to enter and edit text and save and load from disk, as you would expect. It also lets you print some or all of your text, search and replace text patterns and much more. It is Intuition-based, which means it uses all the user-friendly features of the Amiga that you have become familiar with on your computer such as windows, menus and mice. However, if you're a die-hard used to the hostile world of computers before the advent of WIMPs, you'll be pleased to know you can do practically everything you'll want to do from the keyboard without having to touch a mouse.The editor is 'RAM-based', which means that the file you are editing stays in memory for the whole time, so you don't have to wait while your disk grinds away loading different sections of the file as you edit. As all editing operations, including things like searching, are RAM-based they act blindingly quickly.

When you have typed in your programs it is not much use if you are unable to save them to disk, so the editor has a comprehensive range of save and load options, allowing you to save all or part of the text and to load other files into the middle of the current one, for example.

To get things to happen in the editor, there are various methods available to you. Features may be accessed in one or more of the following ways:

- Using a single key, such as a Function or cursor key;
- Clicking on a menu item, such as Save;
- Using a menu shortcut, by holding down the Right Amiga key (subsequently referred to as A) and pressing another key e.g. A F for Find;
- Using the Control key (subsequently referred to as Ctrl) in conjunction with another, such as Ctrl-A for cursor word left;
- Clicking on the screen.

The menu shortcuts have been chosen to be easy and obvious to remember, the cursor key functions are based on those in the AmigaBASIC editor while the Ctrl commands are based on those used in WordStar, and many other compatible editors since.

A Few Words about Requesters

The editor makes extensive use of Requesters, so it is worth recapping how to use them, particularly for entering text. The editor's requesters contain string gadgets and buttons.

String gadgets enable to to enter and edit text, and are depicted by a box containing the text, and with a block indicating the cursor position. Characters may be typed in and corrected using the Backspace, Del and cursor keys. You can clear the whole edit field by pressing A X. You can move the cursor to the beginning by pressing Shift ←, or to the end by pressing Shift →. If there is more than one editable text field in a requester, you can move between them by clicking near them with the mouse.

Buttons may be clicked-on with the mouse and cause the requester to go away. Usually there is a default button, shown by having a double border. Pressing Return on the keyboard is equivalent to clicking on the default button, so long as a string gadget is active.

Some requesters allow only a limited range of characters to be typed into them - such as the Goto Line requester.

The File Requester

The File Requester is used to select file names for the disk input and output facilities of the editor. In its simplest form all you need to do is to click on the file you require and then on the OK button. To cancel the operation click on the Cancel button. At the top of Requester is the *drawer* specification, this determines which disk and sub-directory is displayed and can include wildcards, for example the specification

```
df1:examples/#?.bas
```

will display all files ending in `.bas`.

If you edit this specification, pressing the Return key will cause the directory to be read and displayed in the main part of the requester. Files may be selected by clicking on them then pressing Return or clicking on OK. The file list shows sub-directories with a (dir) prefix and the scroll bar may be used to navigate the file list. Files may be selected or the requester Cancelled while the directory is still being read! When initially invoked, only the first few files will be displayed. To update the file list, click on the slider to the right of the filename list.

You can obtain a list of all devices (e.g. DF0:, DF1:) by right-clicking within the file requester.

This File Requester uses the ARP library by Charlie Heath. If this library is not found then a simple string gadget will be used in its place.

Entering text and Moving the cursor

Having loaded **HiSoft BASIC Professional**, you will be presented with an empty window with a status line at the top and a orange block, which is the cursor, in the top left-hand corner.

The status line contains information about the cursor position in the form of Line and Column offsets as well as the number of bytes of memory which are free to store your text. Initially this is displayed as 9980, as the default text size is 10000 bytes. You may change this default if you wish, together with various other options, by selecting Preferences, described later. The 'missing' 20 bytes are used by the editor for internal information.

The rest of the status line area is used for error messages, which will usually be accompanied by a 'ping' noise to alert you. Any message that gets printed will be removed when subsequently you press a key.

To enter text, you type on the keyboard. As you press a key it will be shown on the screen and the cursor will be advanced along the line. If you are a very good typist you may be able to type faster than the editor can re-display the line; if so, don't worry, as the program will not lose the keystrokes and will catch up when you pause. At the end of each line you press the Return key (or the Enter key on the numeric pad) to start the next line. You can correct your mistakes by pressing the Backspace key, which deletes the character to the left of the cursor, or the Delete key, which removes the character the cursor is over.

The main advantage of a computer editor as opposed to a normal typewriter is its ability to edit things you typed a long time ago. The editor's large range of options allow complete freedom to move around your text at will.

Cursor keys

To move the cursor around the text to correct errors or enter new characters, you use the cursor keys, labelled ← → ↑ and ↓. If you move the cursor past the right-hand end of the line this won't add anything to your text, but if you try to type some text at that point the editor will automatically add the text to the real end of the line. If you type in long lines the window display will scroll sideways if required.

If you cursor up at the top of a window the display will either scroll down if there is a previous line, or print the message Top of file in the status line. Similarly if you cursor down off the bottom of the window the display will either scroll up if there is a following line, or print the message End of file.

For those of you used to WordStar, the keys Ctrl-S, Ctrl-D, Ctrl-E and Ctrl-X work in the same way as the cursor keys.

To move immediately to the start of the current line, press Ctrl ←, and to move to the end of the current line press Ctrl →.

To move the cursor a word to the left, press Shift ← and to move a word to the right press Shift →. You cannot move past the end of a line with Shift →. A word is defined as anything surrounded by a space, a tab or a start or end of line. The keys Ctrl-A and Ctrl-F also move the cursor left and right on a word basis.

To move the cursor a page up, press Ctrl-R or Shift ↑. To move the cursor a page down, press Ctrl-C or Shift ↓.

If you want to move the cursor to a specific position on the screen you may move the mouse pointer to the required place and click (There is no WordStar equivalent for this feature!).

Tab key

The Tab key inserts a special character (ASCII code 9) into the buffer, which on the screen looks like a number of spaces, but is rather different. Pressing Tab aligns the cursor onto the next 'multiple of 8' column, so if you press it at the start of a line (column 1) the cursor moves to the next multiple of 8, +1, which is column 9. Tabs are very useful indeed for making items line up vertically and its main use in **HiSoft BASIC Professional** is for such things as indenting structured program lines. When you delete a tab the line closes up as if a number of spaces had been removed. The advantage of tabs is that they take up only 1 byte of memory, but can show on screen as many more. You can change the tab size before or after loading **HiSoft BASIC Professional**; to change the default use the Preferences command described shortly.

Backspace key

The Backspace key removes the character to the left of the cursor. If you backspace at the very beginning of a line it will remove the 'invisible' carriage return and join the line to the end of the previous line. Backspacing when the cursor is past the end of the line will delete the last character on the line, unless the line is empty in which case it will re-position the cursor on the left of the screen.

Delete key

The Delete key removes the character under the cursor and has no effect if the cursor is past the end of the current line.

Goto a particular line

To move the cursor to a specific line in the text, click on Goto line... from the Options menu, or press A G. A requester will appear, allowing you to enter the required line number. Press Return or click in the OK button to go to the line or click on Cancel to abort the operation. After clicking on OK the cursor will move to the specified line, re-displaying if necessary, or give the error End of file if the line doesn't exist.

Go to top of file

To move to the top of the text, click on Goto Top from the Options menu, or press AT. The screen will be re-drawn if required starting from line 1.

Go to end of file

To move the cursor to the start of the very last line of the text, click on Goto Bottom, or press AB.

Quitting HiSoft BASIC Professional

To leave **HiSoft BASIC Professional** and remove it from memory, click on Quit from the Project menu, or press AQ. If changes have been made to the text which have not been saved to disk, an alert box will appear asking for confirmation. Clicking on Cancel will return you to the editor, while clicking on OK will discard the changes and return you to the CLI or Workbench.

Deleting text

Delete line

The current line can be deleted from the text by pressing Ctrl-Y.

Delete to end of line

The text from the cursor position to the end of the current line can be deleted by pressing Ctrl-Q. (This is equivalent to the WordStar sequence Ctrl-Q Y).

UnDelete Line

When a line is deleted using either of the above commands it is preserved in an internal buffer, and can be re-inserted into the text by pressing Ctrl-U. This can be done as many times as required, particularly useful for repeating similar lines or swapping individual lines over.

Delete all the text

To clear out the current text, click on New from the Project menu. If you have made any changes to the text that have not been saved onto disk, a confirmation is required and the requisite alert box will appear. Clicking on OK will delete the text, or Cancel will abort the operation.

Disk Operations

It is no use being able to type in text if you are unable to save it anywhere permanently, or load it back subsequently, so the editor has a comprehensive set of features to read and write to disk.

Saving Text

To save the text you are currently editing, click on Save As from the Project menu, or press A S. The File Requester (see earlier) will appear, allowing you to select a suitable disk and filename. Clicking OK or pressing `Return` will then save the file onto the disk. If an error occurs a requester will appear showing a suitable error message or an AmigaDOS error number, the exact meaning of which can be found in **Appendix B.**

If you click on Cancel the text will not be saved. Normally if a file exists with the same name it will be deleted and replaced with the new version, but if Backups are selected from the Preferences options then any existing file will be renamed with the extension `.BAK` (deleting any existing `.BAK` file) before the new version is saved.

Save

Save on the Project menu. If you have already done a Save As (or a Load), **HiSoft BASIC Professional** will remember the name of the file and display it in the title bar of the window. If you want to save it without having to bother with the file selector, you can click on Save on the Project menu and it will use the old name and save it as above. If you try to save without having previously specified a filename it will present you with the File Requestor, as in Save As.

Loading Text

To load in a new text file, click on Load from the Project menu, or press A L. If you have made any changes that have not been saved, a confirmation will be required. The File Requester will appear, allowing you to specify the disk and filename. Assuming you do not Cancel, the editor will attempt to load the file. If it will fit, the file is loaded into memory and the window is re-drawn. If it will not fit an alert box will appear warning you, and you should use Preferences to make the edit buffer size larger, then try to load it again.

Note The editor only understands ASCII files; tokenised files, such as those produced by AmigaBASIC have to be de-tokenised first. See **Appendix C** for details of how to do this.

Inserting Text

If you want to read a file from disk and insert it at the current position in your text click on Insert File from the Project menu, or press A I. The File Requester will appear and assuming that you do not cancel, the file will be read from the disk and inserted, memory permitting.

Directory

This command (A O) lets you change the current directory that you are using. This is similar to using the **CHDIR** command in direct mode in AmigaBASIC.

Searching and Replacing Text

To find a particular section of text, click on Find from the Search menu, or press A F. A requester will appear, allowing you to enter the Find and Replace strings. If you click on Cancel no action will be taken. If you click Next (or press Return) the search will start forwards from the current cursor position. Clicking on Previous will start the search backwards. If you do not wish to replace (i.e. simply find things) leave the Replace string empty.

If the search was successful, the screen will be re-drawn at that point with the cursor positioned at the start of the string. If the search string could not be found, the message Not found will appear in the status area and the cursor will remain unmoved. By default the search is always case-independent, so for example if you enter the search string as test you could find the words TEST, Test and test. If you click on the UPPER & lower case Different button the search will be case-dependent.

To find the next occurrence of the string click on Find Next from the Search menu, or press A N. The search starts at the position just past the cursor.

To search for the previous occurrence of the string click on Find Previous from the Search menu, or press A P. The search starts at the position just before the cursor.

Having found an occurrence of the required text, it can be replaced with the Replace string by clicking on Replace from the Search menu, or by pressing A R. Having replaced it, the editor will then search for the next occurrence.

If you wish to replace every occurrence of the find string with the replace string from the cursor position onwards, click on Replace All from the Search menu. During the global replace the Esc key can be used to abort and the status area will show how many replacements were made. There is deliberately no keyboard equivalent for this to prevent it being chosen accidentally.

To search and replace Tab characters just press Tab when typing in the requester. Other control characters may be searched for by typing them in directly (Ctrl-G for example). However do *not* use this for CR (Ctrl-M) and LF (Ctrl-J) characters.

Block Commands

Marking a block

The start of a block is marked by moving the cursor to the required place and pressing key F1. The end of a block is marked by moving the cursor and pressing key F2. The start and end of a block do not have to be marked in a specific order - if it is more convenient you may mark the end of the block first.

Saving a block

Once a block has been marked, it can be saved by pressing key F3. If no block is marked, the message What blocks! will appear. If the start of the block is textually after its end the message Invalid block! will appear. Both errors abort the command. Assuming a valid block has been marked, the File Requester will appear, allowing you to select a suitable disk and filename. If you save the block with a name that already exists the old version will be overwritten - no backups are made with this command.

Copying a block

A marked block may be copied, memory permitting, to another part of the text by moving the cursor to where you want the block copied and pressing key F4. If you try to copy a block into a part of itself, the message Invalid block will appear and the copy will be aborted.

Deleting a block

A marked block may be deleted from the text by pressing Shift-F3 or Shift-F5. The shift key is deliberately required to prevent it being used accidentally. A deleted block is remembered, memory permitting, in the *block buffer*, for later use.

Copy block to block buffer

The current marked block may be copied to the block buffer, memory permitting, by pressing Shift-F4. This can be very useful for moving blocks of text between different files by loading the first, marking a block, copying it to the block buffer then loading the other file and pasting the block buffer into it.

Pasting a block

A block in the block buffer may be pasted at the current cursor position by pressing F5.

The block buffer will be lost if the edit buffer size is changed or a compile occurs.

Printing a block

A marked block may be sent to the printer by clicking on Print Block from the Project menu, or by pressing A W. A simple requester will appear asking for the name of the printer, which defaults to PRT:, and clicking on OK will print the block. Of course the name can be any valid AmigaDOS device, so you could 'print' the block to disk if required. It is different to Save Block in that tabs are expanded to spaces.

If you try to Print when no block is marked at all then the whole file will be printed.

Miscellaneous Commands

Help Screen

The key equivalents for the commands not found in menus can be seen by pressing the Help key, or A H. A requester will appear showing the function keys, as well as the free memory left for the system. The number in brackets shows the size of the largest free block of system memory.

Preferences

Selecting Preferences from the Options menu will produce a requester allowing you to change several editor settings:

Tab setting

By default, the tab setting is 8, but this may be changed to any value from 2 to 16.

Text Buffer

By default the text buffer size is 10000 bytes, but this can be changed from 4000 to 999000 bytes. This determines the largest file size that can be loaded and edited. Care should be taken to leave sufficient room in memory for compilations - pressing the Help key displays free system memory, and for compilations this should always be at least 100k bytes. Changing the editor workspace size will cause any text you are currently editing to be lost, so a confirmation is required if it has not been saved.

Backups

By default the editor does not make backups of programs when you save them, but this can be turned on by clicking on the Yes button. Backup files do not have any icon associated with them.

Auto Indent

It can be particularly useful when editing programs to indent subsequent lines from the left, so the editor supports an auto-indent mode. When active, an indent is added to the start of each new line created when you press Return. The contents of the indent of the new line is taken from the white space (i.e. tabs and/or spaces) at the start of the previous line. Click on YES if you want this feature.

Auto full-size

By default, the editor will not use the entire screen display for compatibility with US Amigas. Click on YES in the Preferences requester to make the editor use a large a window as possible. We recommend this be avoided by A500 users with PAL screens as this uses a surprising amount of extra memory.

Saving Preferences

If you click on the Cancel button any changes you make will be ignored. If you click on the OK button the changes specified will remain in force until you quit the editor. If you would like the configuration made permanent then click on the Save button, which will create the file HBASIC.INF on your disk. Next time you run **HiSoft BASIC Professional** the configuration will be read from that file.

In addition to saving the editor configuration the current program buffer size, shown in the compilation options requestor, is also saved.

Compiling and Running Programs

All compilation and run options can be found on the Program menu.

Compiling

To set all compilation options and compile the program you are currently editing click on Compile from the Program menu or press A C, and a large requester will appear allowing you to set all the options.

The exact differences between most of the code generating options are explained in more detail in **Appendix A**. The only option covered here is Compile to. **HiSoft BASIC Professional** can compile to disk or to memory - compiling to memory is much faster and ideal for trying things out quickly, while compiling to disk means you can create stand-alone, double-clickable programs without any need for the compiler to be present.

If you haven't saved your program when you compile to disk, the file will have the name NONAME on the disk.

The file produced on the disk can then be run from the Workbench directly by double-clicking on it like any other program. It does not need any of the compiler or its library file to be present - it is completely self contained.

After you click on OK or press Return the compilation process will start, described more fully in the next chapter. At the end of the compilation the program will wait for a key press, allowing you to read any messages produced, before returning you to the editor. If there were any compilation errors the editor will go to the first erroneous line and display the error message in the status bar. Subsequent errors may be investigated by pressing A J.

Note If you compile or run a program on a machine with less than 200k free (this normally means you have less than 1M of RAM in total), the editor's window is reduced to its minimum size, to save memory. It will return to its full size after the compilation or when your program finishes

Running Programs

If you click on Run from the Program menu or press A X you can then run a program previously compiled into memory. When your compiled program finishes it will normally wait for a key then return you to the editor. If you have made any changes to your program since the last compilation, then an automatic compilation will occur, before the program is run. If there are any compilation errors then it is not possible to run the program. When a compilation occurs as a result of a Run command it uses the current options as set by the last Compile command.

Programs compiled to memory then run using this command work in a very similar way to when they are compiled to disk and double-clicked, though there is one important difference - they have much less memory available when they run, as an editor and compiler are sharing the memory space with them.

Jump to Error

During a compilation any errors that occur are remembered, and can be recalled from the editor. Clicking on Jump to error from the Program menu, or pressing A J will move the cursor to the next line in your program which has an error, and display the message in the status line of the window. You can step to the next one by pressing A J again, and so on, letting you correct errors quickly and easily. If there are no further errors when you select this option the message No more errors will appear, or if there are no errors at all the message What errors! will appear.

Window Usage

The window used by the editor works like all other Intuition windows, so you can move it around by using the Move bar on the top of it, you can change its size by dragging on the size gadget. Clicking on the Close gadget is equivalent to choosing Quit from the Project menu.

Automatic Double Clicking

HiSoft BASIC Professional will be loaded automatically whenever a source file is double-clicked from the Workbench, so long as its Default Tool setting under the Info item on the Workbench menu is set correctly.

Chapter 4
Concepts

This chapter describes the technical details of the **HiSoft BASIC Professional** language, together with some of its more advanced features. It is intended for users who already have a good understanding of the BASIC language and want to get to grips with **HiSoft BASIC Professional** quickly. If you are new to BASIC please read the tutorial first.

Character Set

HiSoft BASIC Professional uses plain ASCII characters in its input files. The following characters have special meanings:

a-z, A-Z The letters, which are used in reserved words and the user's variable names, labels and sub-program names. Lower and upper case are treated as the same in variable and reserved word definitions so that THEN, then and Then are all the same reserved word.

E, e, D, d are also used for exponents in numbers.

0-9 The digits, which are used in numbers and can also be used in names as long as they are not the first character.

. The full stop or period, which is used as the decimal point in numbers and can also be used in names as long as it is not the first character.

% The percentage sign, which is used to indicate that a variable is a 16 bit integer i.e. whose values must be in the range **-32768** to **32767**.

& The ampersand, which is used to indicate that variables are long integers i.e. whose values must be in the range **-2^31** to **2^31-1**. Also used to introduce hexadecimal, octal and binary constants.

! The exclamation mark, which is used to indicate that a variable is a single-precision floating point number.

#	The hash or number sign, which is used to indicate that a variable is a double precision floating point number and also used to indicate that certain input/output operations are to be directed to channels rather than the screen (e.g. `PRINT #`).
$	used to indicate string variables.
_	The underline character, which can be used in variables after the first character assuming the underline (**U**) flag has not been turned off. If it appears at the start of a symbol or the underline (**U**) has been disabled then it indicates the rest of the line is to be ignored and that the following line is to be considered part of the current one.
"	The quotation mark or double quote which is used to delimit string literals.
'	The apostrophe or single quote which is used to indicate that the rest of this line is to be regarded as a comment.
()	The parentheses or round brackets, which are used to enclose function arguments, over-ride the priority of operators and indicate arrays.
+ - * /	the basic arithmetic operators.
=	assignment operator and equality operator.
< >	less than and greater than comparison operators.
^	exponentiation operator.
\	The back-slash character, which is used as the integer division operator.
,	comma.
;	semi-colon.
?	used as an abbreviation for `PRINT`.

Other characters with ASCII values lower than 32 are treated as white space, and ignored so you may, for example, include form-feed (**chr$(12)**) characters to give a new page on your printer when listed.

Other characters may be used in strings, but otherwise will generate a warning and will be ignored.

Program lines and labels

Program lines consist of an optional line number or label, one or more statements separated by colons and an optional comment, which starts with an apostrophe or single quote.

Line numbers may be any number between **1** and **65529** inclusive. (**65529** may seem a strange number, it is the maximum allowed by Microsoft BASIC).

Line labels consist of any valid variable name that is not used as a variable or a sub-program and labels are followed by a colon. There is no limit to the number of characters in a line label but lower- and upper-case letters are treated as the same and the characters must not be a reserved word. Thus the following line labels are allowed:

```
Label9999:

A.very.long.label.that.causes.problems.to.other.BASICs:

Hello:
```

The following line labels are the same:

```
Start:

START:

start:
```

Line numbers and labels may be preceded by white space. White space is not required after the last digit or colon. Line numbers may not contain spaces.

Note: For compatibility with other BASICs we recommend that you do not use full stops (.) or underlines (_) in labels and keep them to less than 40 characters.

Line number **0** is not allowed because it would be confused when using `ON ERROR GOTO 0` which does not mean go to line **0** if an error occurs.

In general we do not recommend the use of line numbers since line labels are much more readable. The exception to this is when using **ERL** when the use of line numbers is essential.

Most of the time you do not need to use line numbers or line labels because **HiSoft BASIC Professional** has such a rich set of structured statements. (Much better than Pascal and even more flexible than C and Modula 2.)

You may have many statements per line provided each is separated by a colon.

HiSoft BASIC Professional has an extension to call sub-programs without the **CALL** keyword. However you cannot do this if the sub-program has no parameters and is the first statement of a multi-statement line. For example, if you have:

```
SUB john STATIC

PRINT "John";

END SUB
```

then

```
john

PRINT " Smith"
```

will print

```
John Smith
```

as will

```
call john: print " Smith"
```

However

```
john: print " Smith"
```

is wrong because the `john:` could be a label definition; normally the compiler will warn you on the parsing phase if you make this error.

This problem does not apply if the sub-program has a parameter, for example:

```
SUB john(para$) static

PRINT "John ";para$;

END SUB

john "David": print " Smith"
```

is fine because it cannot be mistaken for a label.

If an apostrophe or single quote (') appears on a line then the rest of the line is treated as comment and ignored. The only exception to this is **DATA** statements which treat the apostrophe as part of the data. If you want a comment on such a line precede it with a colon; this will terminate the **DATA** statement.

Program lines may be any length theoretically, but it is generally a good idea to keep them less than 80 characters so that the whole line is displayed at once. If you need a line that is significantly longer than this then the chances are that the line is more complicated than it should be as far as ease of understanding is concerned. The exception to this is the **FIELD** statement where for large records you need many more than 80 characters.

To get round this the underline character (_) may be used to cause lines to be continued on the next physical line. Anything after the underline is ignored. For example:

```
FIELD #3,20 AS name$, _ ' surname only
        5 AS initials$,_
        50 AS street$, _' include name or number here
        20 AS town$,_
        20 AS county$,_ ' or state if applicable
        20 AS country$
```

which would be much more readable than the one-line equivalent where you would also have to leave out the comments.

Normally **HiSoft BASIC Professional** allows underlines in variable names, unlike traditional BASICs. Underlines are treated as continuation characters if they are not part of a variable name. If you are porting programs that have continuation characters immediately after identifiers or reserved words then use the **U-** option or click on the appropriate box in the **Compile** requester.

For example:

```
IF x THEN_
      PRINT "hello"
```

will be accepted in some BASICs as a one-line `IF` statement (no need for an `END IF`). Without the **U-** flag **HiSoft BASIC Professional** will give an error because it thinks you are using a variable called `THEN_`. To solve this use **U+** or insert a space in front of the _ character.

If you use **U-** and inadvertently use an identifier containing an underline you may get a very strange error message because the compiler has ignored the rest of the line.

Data Types

There are five types of data in **HiSoft BASIC Professional:**

Strings

A string is a sequence of characters that may be up to 16 megabytes long assuming you have enough memory. Strings may contain any character with a value of 0 to 255 inclusive.

Integers

Integers are numeric and consist of the whole numbers from **-32768** to **32767**.

Long Integers

Long integers are numeric and consist of the whole numbers between **-2147483648** and **2147483647**.

Single precision numbers

Single-precision numbers have approximately seven digits of precision and a range of **5.4E-20** to **9.2E-18** for positive values and **-2.7E-20** to **-9.2E18** for negative values.

Double precision numbers

Double precision numbers have approximately 16 digits of precision and a range of **4.9E-324** to **1.8E308** for positive numbers and **-4.9E-324** to **-1.8E308** for negative numbers. There is loss of precision with numbers of magnitude less than **2.2E-308**.

Constants

Constants are values which do not change during program execution. Constants may be of all 5 types.

A string constant is a sequence of ASCII characters enclosed in double quotes ("). These can be any character between **32** (space) and **255**. To obtain a double-quote in a string repeat it, so that, for example, the string consisting of one double quote character is """". The first is the start of the string, the second and third form the character itself and the last is the closing quote.

Numeric constants are formed in one of the following ways:

Decimal numbers

A sequence of decimal digits followed optionally by a decimal point (.) and more digits and/or an exponent. An exponent consists of the letter **d**, **D**, **e** or **E** followed by a decimal integer. **E** indicates single precision and **D** indicates double precision. The number may be preceded by a minus sign as may the numeric part of the exponent. The number before the decimal point may be omitted. The number may be followed by a type specifier (**%**,**!**,**&** or **#**).

Hexadecimal Constants

Hexadecimal constants start with **&H** or **&h** and are followed by hexadecimal digits(**0**-**9**, **a**-**f**, **A**-**F**). The number may be followed by a type specifier (**%**,**!**,**&** or **#**).

Hexadecimal integer constants between **&h8000** and **&hFFFF** are taken as signed 16 bit integers. Hexadecimal long integer constants between **&h80000000** and **&hFFFFFFFF** are treated as signed 32 bit constants.

For example:

&h7FFF	= 32767	integer
&h8000	= -32768	integer
&h8001	= -32767	integer
&hFFFF	= -1	integer
&h10000	= 65536	long integer
&h7FFFFFFF	= 2147483647	long integer
&h80000000	= -2147483648	long integer
&hFFFFFFFF	= -1	long integer
&h100000000	= 4294967296	double

If you want **&h8000** to be treated as **+32768** then follow the number with **&** and it will be treated as a long and thus positive.

e.g.

&h8000&	= 32768	long integer

Octal Constants

Octal constants start with **&O** or **&o** or just simply **&**. and are followed by octal digits (**0-7**). The number may be followed by a type specifier (**%,!,&** or **#**). The type of an un-terminated octal constant is determined by the same rules as for hexadecimal constants (see above).

Binary Constants

Binary constants start with **&B** or **&b** and are followed by the digits **0** or **1**.The number may be followed by a type specifier (**%,!,&** or **#**). The type of an un-terminated binary constant is determined by the same rules as for hexadecimal constants described previously.

Character constants

These start like strings of only one character and are followed by the **%** character and have a value equivalent to the **ASC()** of the character. However they are generally easier to read and more efficient than the **ASC()** equivalent.

Types of Constants

The rules regarding what type a constant is are rather complicated but in general you should find that normally the compiler does what you expect. The most common problem is that some hexadecimal constants are treated as negative. If this is a problem please see the **Hexadecimal Constants** section above. The following are in **decreasing** order of importance:

1. A terminating character is used. If the number is terminated by:

%	it is taken as an integer
&	it is taken as a long integer
!	as a single precision floating point number
#	as a double precision floating point number.

2. If the number is hexadecimal or octal and lies in the following range:

0 to **&hFFFF**	it is taken as an integer
&h10000 to **&hFFFFFFFF**	it is taken as a long integer
&h100000000 upwards	it is taken as a double.

For the rules concerning whether a constant is treated as negative see above.

3. If the number is decimal and it is not a whole number and has *more* than **6** digits in the whole number and decimal parts, then it is a double.

4. If it has an exponent of the form **D** or **d** then it is a double.

5. If it has an exponent of the form **E** or **e** then it is an integer.

6. If it has a decimal point then it is a single precision number.

7. If it is a whole number less than or equal to **32767** it is an integer.

8. If it is a whole number less than or equal to **2147483647** it is a long integer.

Examples:

`1`	integer
`1.0`	single precision (.)
`1.0E0`	single precision(. and E)
`1E1`	single precision
`1.00000`	single precision
`1.000000`	double precision (7 digits)
`1.000000E0`	single precision(E overides 7 digits)
`1D0`	double precision (D)
`1D0%`	integer (% over-rides D)
`1.0&`	long integer (& overides .)
`1D0!`	single precision (! overides D)
`1#`	double precision

Variables and Reserved Words

Variable names start with a letter and subsequent characters may be letters, digits, or full stops (.). In addition underlines (_) may be included if you haven't switched this off using the **U-** option. For maximum compatibility with other BASICs don't use underlines or full stops.

Lower and upper case are treated as the same in variable names and reserved words so that `PRINT` , `Print` and `Print` are all the same reserved word.

Variables may be terminated with a type specifier **%** (integer), **&** (long integer), **!** (single precision floating point) or **#** (double precision floating point). If there is no type specifier then the type is determined by the current **DEFtype** statement for the first letter of the variable. If there have been no **DEFtype** statements then single precision (!) is used.

Compiler error messages specifying variable names always include the type specifier that has been assumed.

For example, the following gives the types of the respective variables:

```
DEFINT i-k
DEFSTR s
DEFDBL q-r
```

`i%`	integer
`i`	integer `i%` (same as above)
`I`	(also the same as above)
`i&`	long integer (different)
`str1`	string (same as `str1$`)
`real_value1`	double (same as `real_value1#`)

You can not use reserved words as the names of variables or sub-programs. The reserved words are listed in full in **Appendix D**. Reserved words and variables may be entered in upper or lower case or a mixture of both.

In general using reserved words with type specifiers should be avoided for compatibility reasons.

GO is not a reserved word. However if **GO** is followed by **TO** or **SUB** then it is made into **GOTO** or **GOSUB** respectively; so you can have white space between **GO** and **TO** and it will still be treated as **GOTO**. Thus you can use **GO** as a variable name if you like but some strange things can happen, such as

```
FOR i=go TO from STEP 2
```

is misunderstood because the compiler considers this to be

```
FOR I = GOTO  from  STEP 2
```

Variables must not start with **FN** because they would be treated as function names. The same rules for determining the type of a variable are used to determine the types of functions. The following are function names:

```
FNtest

FNsine&

FNget.one.character
```

Sub-program names must not have a type specifier because they do not have a type. You may use the same name for a variable and a sub-program although the type specifier of the variable must be used explicitly. However, this can be *very* confusing. For example

```
SUB john

john%=42

END SUB

john%=52: john
```

Arrays

Array names follow the same rules as for variables and their types are determined in the same way. You may use the same name for an array and an ordinary variable. Normally arrays are followed by an open parenthesis, except in the **ERASE** statement and the **UBOUND** and **LBOUND** functions when this is assumed automatically.

Arrays are tables of values each of the same type. Normally the number of elements in an array and the number of dimensions is specified with a **DIM** statement (see **Chapter 5**). There are no restrictions on the size of arrays other than available memory and subscripts may be long integer values if applicable. The maximum number of dimensions for an array is 31 (which would take up a minimum of 4 gigabytes of memory if each index had more than one element).

If the **DIM** statement is not used the maximum subscript is assumed to be 10. If you have switched off array checks using compiler option **A-** then you *must* use the **DIM** statement. The minimum value of subscripts is 0 unless an **OPTION BASE** statement is used. When referenced, the element of the array to be accessed is specified by one or more expressions inside parentheses and separated by commas. The expressions may be of any numeric type although single and double precision real values will be converted to long integers.

For example, given:

```
DIM A(30), B$(table_entries,4), table&(100000)
DIM t%(fred*fred),c(n,n,n)
```

then the following are valid array references:

```
A(i)
B$(j*3,2)
table&(i&)
t%(k-1)
c(i,j,k)
```

For more information on arrays see the **Advanced Arrays** section in this chapter.

Operators

Expressions are made up of constants, variables, array variables, function calls and operators. The order of priority is listed below with the highest priority first:

1. Exponentiation (to the power of) (**^**)
2. Unary Minus (**-**)
3. Multiplication (*****) and Floating Point Division (**/**)
4. Integer Division (****)
5. Modulus (**MOD**)
6. Addition (**+**) and Subtraction (**-**)
7. Comparisons (**=,<>,>,<,>=,<=, ==**)
8. **NOT**
9. **AND**
10. **OR** and **XOR** (exclusive or)
11. **EQV**
12. **IMP**

The only exception to this is that **x^-y** is evaluated as **x^(-y)**.

To change the order of evaluation use parentheses (round brackets).

The guiding principle for the precision used when evaluating expressions is that the minimum precision is used that will ensure that accuracy is not lost.

The exponentiation operator (^) always has its operands converted to either single or double precision floating point and returns a result of the same type. Single precision is used if the operands are either integer or single precision. If either operand is a long integer or is double precision then it is evaluated in double precision for accuracy. See **Figure 4-1** below. This operator uses logarithms to give its result and as such is slow and inaccurate if the second operand is a small integer.

The multiplication, addition, subtraction and unary minus operators may have operands of any numeric type with the following table giving the result of the expressions:

	integer	**long**	**single**	**double**
integer	integer	long	single	double
long	long	long	double	double
single	single	double	single	double
double	double	double	double	double

Figure 4-1

Addition may be also used for strings when it means concatenation so that, for example:

```
"ABC" +"DEF" ="ABCDEF"
```

Floating point division operands are always converted to single or double precision floating point numbers, the following table (**Figure 4-2**) gives the result of the expression:

	integer	**long**	**single**	**double**
integer	single	double	single	double
long	double	double	double	double
single	single	double	single	double
double	double	double	double	double

Figure 4-2

The integer division operator \ uses long integer (32-bit) arithmetic *unless* both operands are integers in which case integer (16-bit) arithmetic is used.

The comparison operators always return an integer value of -1 for true and 0 for false. The comparison is evaluated using the type given in **Figure 4-1** above for numeric types. Strings may also be compared.

The comparison operators are

=	equality
<>	inequality
>	greater than
<	less than
>=	greater or equals
<=	less than or equals
==	almost equals (two equals signs)

The almost-equals operator is a **HiSoft BASIC Professional** extension for single or double precision floating point comparisons and it is defined as follows:

```
x==y
```

calculates

```
ABS(x-y)  <=  ABS(y  *  1E-6)
```

Thus == can be used to check for near equality even if a small number of rounding errors have been introduced. For integers and long integers the comparison is the same as equals and for strings the comparison is the same as equals except that lower case letters are treated as equal to their uppercase counterparts. For example:

`2.0==2.0`	is true
`2.0==1.999999`	is true
`2.0==1.99999`	is false

A string is considered less than another if the first character that differs is less in the first string. If the strings are the same until one string is exhausted then the shorter string is less. All the following examples are true:

`"Fred"<"Hello"`	because `"F"<"H"`
`"Frederick"<"Hello"`	because `"F"<"H"`
`"fred">"Hello"`	because `"f">"H"`. The lower case letters come after the upper.
`"Frederick">"Fred"`	because `"Frederick"` is longer

All the logical operators **NOT**, **AND**, **OR**, **XOR**, **EQV** and **IMP** use long integer arithmetic (32-bit) unless both operands are integers in which case integer arithmetic is used. These operators work bitwise, with each bit affected as shown below.

X	Y	NOT	AND	OR	XOR	IMP	EQV
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	0	0
1	1	0	1	1	0	1	1

Although these operations work on the individual bits they have the same affect as the corresponding logical operators if you use -1 for TRUE and 0 for FALSE.

Examples:

```
-1 OR -1        =-1

4 OR 3          = 7     (100 and 011 in binary)

-1 XOR 0        =-1

8 AND 4         = 0     (1000 and 100 in binary)
```

Sub-programs and User Defined Functions

Sub-programs and user defined functions are one of the most powerful features of many modern BASICs and **HiSoft BASIC Professional** takes these ideas even further.

The idea of a sub-program is to isolate part of the code of your program in a way that makes it easy to call and easy to ensure that it is not interfering with variables that it isn't supposed to use.

The simplest definition of a sub-program is something like

```
SUB hello
PRINT "hello"
END SUB
```

The **SUB** statement defines the name of the sub-program that we are defining and the **END SUB** indicates that we have finished.

Sub-program definitions may not contain other sub-program definitions.

The `hello` sub-program can be called using

```
call hello
```

or even just

```
hello
```

and will print the word `hello`. You can call sub-programs before or after their declarations.

So far this doesn't give us anything that you can't do with old-style BASIC **GOSUB...RETURN** statements. However by passing parameters to sub-programs we can make the sub-program work on different variables or values.

Variable Parameters

Sub-programs may have two different sorts of parameters, value and variable parameters. By default parameters are variable parameters and are passed by *reference*. This means that if the sub-program modifies the parameter the variable that it is called with is modified. For example:

```
SUB TimesTwo(v)
v=v*2
END SUB
```

If we call this using

```
INPUT " Enter a number";i
CALL TimesTwo(i)
PRINT i
```

and enter the number `42`, `i` will be modified and then twice this, `84` will be printed.

The shortened form of the `CALL` statement above is:

```
TimesTwo i
```

Note that the brackets are not used.

When using variable parameters, if you pass an expression rather than a variable of the required type then any modifications to the parameter are lost. Thus the type of the variable *must* be the same as the parameter.

If we changed the calling code to be:

```
INPUT i#
TimesTwo i#
PRINT i#
```

then the variable `i#` would not be modified.

You can pass array elements as variable parameters; this causes the subscripting expression to be calculated before the sub-program is called.

e.g.

```
TimesTwo a(3)
```

would double the value of `a(3)`. However this should be avoided if you are using **ERASE** and **REDIM APPEND** inside sub-programs; see the **Advanced Arrays** section in this chapter for more information.

If you want to call a sub-program that normally would modify the variable, but on this occasion you don't want this to happen, then enclose the variable name in parentheses e.g.

```
TimesTwo (i)
```

If you have more than one parameter for a sub-program then they should be separated by commas in both the call and the definition.

For example:

```
SUB Multiply(i,j,k)
k=i*j
END SUB

Multiply 2,3,i
PRINT i
```

This will print `6`. Note that the `i` that is a parameter and used in the sub-program is an entirely different entity to the `i` in the main program.

Value Parameters

Parameters may also be called by *value*, which means that a variable will not be modified. To indicate that a parameter is passed by value precede it in the definition with the keyword **VAL**. So the above example could be coded as:

```
SUB Multiply( VAL i,VAL j,k)
k=i*j
END SUB

Multiply 2,3,i
PRINT i
```

Value parameters are more efficient than variable parameters and are a **HiSoft BASIC Professional** extension. In most other BASICs with sub-programs you must use variable parameters and enclose them in parentheses. This works fine unless you forget the brackets, when you can modify your main program variables by mistake. In general make a parameter a **VAL** parameter unless you want to return a value.

STATIC variables

In the examples so far we have only used parameters inside sub-programs. However sub-programs may have their own variables. For example

```
SUB Sum(val n, k)
STATIC count,total
total=0
FOR count=1 TO n
        total=total+count
NEXT count
k=count
END SUB

Sum 4,result

PRINT result
```

will print

```
10
```

which is 1+2+3+4.

The word **STATIC** is used to introduce ordinary local variables. You can use commas to separate them. In fact if you omit the **STATIC** statement, the above will still work because **STATIC** is assumed by the compiler. However we recommend strongly that you use this statement together with the variable checks flag **(V+)** as described in **Appendix A**. This will warn you if you mis-spell variables in sub-programs.

For example if we had typed

```
k=k+cont
```

in the example above, the compiler would complain that the variable `cont` was not declared.

STATIC variables are zeroed when the program starts running but are not modified between different calls to the procedure. In the example above if we called the `Sum` sub-program again `Total` would have a value of `10` so we must zero it each time.

SHARED variables

You can also use variables from your main program inside sub-programs by using the **SHARED** statement. For example we could code the example above as:

```
SUB Sum(VAL n)
STATIC count,total
SHARED k
total=0
FOR count=1 TO n
        total=total+count
NEXT count
k=count
END SUB

Sum 4

PRINT k
```

This is however less flexible than the original example because it modifies only one particular variable.

Using **SHARED** variables with variable parameters which should be value parameters can lead to the following difficult-to-spot bug shown on the next page:

```
SUB process(t)
SHARED token
IF t=3 OR t=4 THEN
        .
        .
        token=5
        .
        .
        IF t=4 THEN     ' problem
        .
        .
        END IF
END IF
```

One would naturally expect `t` to be `3` or `4` at the point marked `problem` since `t` was `3` or `4` in the previous `IF` statement. However if the sub-program `process` was called as

```
process token
```

then this would not be the case because the modification of `token` will also change `t`. This can be solved by enclosing `t` in parentheses or, even better, by making the parameter a value parameter. This problem can also occur if the variable token was modified by a sub-program that is called inside `process`, which is even more difficult to spot.

If you have some variables that are imported into many sub-programs and you wish to avoid having **SHARED** statements each time, you can use the **DIM SHARED** statement which causes the variable to be **SHARED** with every sub-program. For example, if you have

```
DIM SHARED debug_flag
```

then you can use `debug_flag` anywhere in your program.

Recursion and Local variables

Sub-programs may be called recursively i.e. they may call themselves.

```
SUB Fibonacci(VAL n, r)
LOCAL temp1,temp2

SELECT CASE n
CASE 0: r=0
CASE 1: r=1
CASE REMAINDER:
        Fibonacci n-1, temp1
        Fibonacci n-2, temp2
        r=temp1*temp2
END SELECT

END SUB

FOR i=0 TO 15
        Fibonacci i,res
        PRINT res;
NEXT i
```

This prints the first few numbers in the Fibonacci sequence, in which the *n*th term is the sum of the two previous terms with the sequence starting with 0, 1, ... This is, in fact not the most efficient way to code this algorithm in **HiSoft BASIC Professional**; the algorithm can also be improved very easily.

The above example also introduces **LOCAL** variables. These are like **STATIC** variables in that they cannot be accessed outside the sub-program. However a new variable is created for each invocation of the sub-program. This becomes important when you have recursive calls. In the example above if there was only one variable `temp1` then it would be corrupted during the second recursive call. Try it and see.

The memory for use of local scalar numeric variables is allocated on the machine stack. If you make heavy use of recursive calls with large numbers of local variables it is possible to run out of stack. Use the **R** option, see **Appendix A** for details.

Strings may also be used as parameters and local variables in exactly the same way as numbers. The only difference is that the actual data in the strings is allocated on the heap and not on the machine stack.

User-Defined Functions

As well as sub-programs you can also have user-defined functions.In **HiSoft BASIC Professional** there are two methods of defining user-defined functions, **DEF FN** and **FUNCTION**. In old-fashioned BASICs and even in **HiSoft BASIC Professional** for the Atari ST at the time of writing only the **DEF FN** form can be used and we will discuss this first. Old-style BASICs restrict you further in that these can only be one line long. For example,

```
DEF FNrad(x)=x*3.141592653589793
```

which converts an angle in radians to degrees and could be used as follows

```
PRINT SIN(FNrad(45))
```

to give the sine of 45 degrees. The names of such user-defined functions must start with `FN`.

However, in **HiSoft BASIC Professional, DEF FN** functions may have all the facilities of sub-programs with the following differences:

User defined functions return results by assigning to a pseudo-variable with the name of the function. For example

```
DEF FNfactorial(n)
IF n<=1 THEN
        FNfactorial=1
ELSE
        FNfactorial=n*FNfactorial(n-1)
END IF
END DEF
```

which calculates the famous factorial function. Note that the definition finishes with `END DEF` and that on the right hand side of the assignment `FNfactorial` causes the function to be called again recursively.

The big difference between user-defined functions and sub-programs is that, in functions, parameters are call-by-value by default and to specify call-by-variable you should precede them with **VARPTR**. If you do not use variable checks, variables are assumed to be **SHARED** rather than **STATIC**.

Here is another coding of the Fibonacci example:

```
DEF FNfibonacci( n)
SELECT CASE n
CASE 0: FNfibonacci=0
CASE 1: FNfibonacci=1
CASE REMAINDER:
        FNfibonacci:= FNfibonacci(n-1)+FNfibonacci(n-2)
END SELECT
END DEF

FOR i=0 TO 15
        PRINT FNfibonacci (i);
NEXT i
```

Clearly this gives a neater solution without the **LOCAL** variables.

Incidentally you can leave out the space between the **FN** and the function name,

The **FUNCTION** syntax makes user-defined functions even more like sub-programs. There is no restriction on the name of the function and the rules for parameters and local variables are the same as for sub-programs. For example, here's yet another coding of the Fibonacci example:

```
FUNCTION Fibonacci(VAL n)
SELECT CASE n
CASE 0: Fibonacci=0
CASE 1: Fibonacci=1
CASE REMAINDER:
        Fibonacci:= Fibonacci(n-1)+Fibonacci(n-2)
END SELECT
END FUNCTION

FOR i=0 TO 15
        PRINT Fibonacci (i);
NEXT i
```

This gives probably the neatest solution to this classic problem.

FUNCTIONs must be declared before they are used. The normal way to this is to ensure that the **FUNCTION END FUNCTION** statements are before any calls of the function. If you wish to use a function before you define it then you can use the **DECLARE** statement. This specifies the parameters of a function in the same way as a **FUNCTION** statement but does not actually contain any code.

For example,

```
DECLARE FUNCTION Fibonacci(VAL n)

FOR i=0 TO 15
        PRINT Fibonacci (i);
NEXT i

FUNCTION Fibonacci(VAL n)
SELECT CASE n
CASE 0: Fibonacci=0
CASE 1: Fibonacci=1
CASE REMAINDER:
        Fibonacci:= Fibonacci(n-1)+Fibonacci(n-2)
END SELECT
END FUNCTION

FOR i=0 TO 15
        PRINT Fibonacci (i);
NEXT i
```

This is our final Fibonacci example (honestly!). A couple of points to note:

•The form of the **DECLARE** statement is exactly the same as the **FUNCTION** statement with the word **DECLARE** at the front.

•**DECLARE** statements are needed for two reasons. First, they enable the compiler to check that the correct number and type of parameters have been used. Second, otherwise the compiler would think that `Fibonacci(i)` in the example above was referring to an array `Fibonacci()`. In fact probably the only advantage of the **FN** syntax is that you can instantly see the difference between a function call and an array access. Of course, you could easily decide to use the same conventions with **FUNCTION** definitions.

DECLARE statements can also be used for sub-programs. They are not required by **HiSoft BASIC Professional** but can be useful for compatibility with other modern BASICs such as QuickBASIC. You can also use them as a documentation aid by having **DECLARE** statements at the front of your program for all the sub-programs and functions .

If you call a function which performs input/output inside another statement that performs input/output strange things may happen. There is no good reason for doing this and it should be avoided.

Arrays and Sub-programs

Arrays may be used as parameters to sub-programs and user-defined functions. They are specified both in call statements and definitions with open and close parentheses after their names. The definition should contain the number of dimensions of the array. Arrays are always passed by reference.

```
DIM b(3,6)
MatSum b(),res

PRINT res

SUB MatSum(a(2),x)
STATIC i,j,x
x=0
FOR i=LBOUND(a,1) TO UBOUND(a,1)
        FOR j=LBOUND(a,2) TO UBOUND(a,2)
                x=x+a(i,j)
        NEXT j
NEXT i

END SUB
```

This sums all the elements of the two dimensional array.The corresponding **DEF** function definition would be:

```
DIM b(3,6)
PRINT fnMatSum( b())

DEF fnMatSum(a(2))
STATIC i,j,x
x=0
FOR i=LBOUND(a,1) TO UBOUND(a,1)
        FOR j=LBOUND(a,2) TO UBOUND(a,2)
                x=x+a(i,j)
        NEXT j
NEXT i
fnMatSum=x
END SUB
```

Sub-programs may share arrays with the main program. The **SHARED** and **DIM SHARED** statements may be used as for scalar variables. The **DIM SHARED** statement when used with arrays also dimensions them. The **SHARED** variables statement should specify the number of dimensions of the array although this is not enforced.

For example,

```
DIM SHARED table(100)
'table() can now be access anywhere in the program.
```

or alternatively

```
DIM table(10)
table(10)=42 : Silly

SUB Silly
SHARED table(1) : PRINT table(10)
END SUB
```

This will print 42.

Local Arrays

Arrays may also be local to a sub-program and both **STATIC** and **LOCAL** varieties are supported. When using **STATIC** you need to make sure that the array is not dimensioned more than once. In the **STATIC** statement the number of dimensions may be included in parentheses. For example,

```
'constants for Table Handler operations
CONST init=0, insert=1, find =2, replace=3

TableHandler init,0,0   'initialise the table
SUB TableHandler(operation, index, value)
STATIC table(1), first_free

SELECT CASE operation
CASE init
        DIM table(100)
        first_free=0

CASE insert
        .
        .
```

In this example the array will only be dimensioned once, as long as the `TableHandler` sub-program is not called with a parameter of `init` more than once.

Using the **LOCAL** statement arrays may be created for the duration of this call to the sub-program. They are erased automatically at the end of the call. The actual dimensions are given in the **LOCAL** statement.

For example:

```
SUB Recursive

LOCAL temp(40)  ' a temporary array with elements up to
temp(40).
.
.
END SUB
```

Advanced Arrays

As well as the **DIM**, **SHARED**, **STATIC** and **LOCAL** statements described above there are a number of other array facilities that are not available in primitive BASICs.

The **UBOUND** and **LBOUND** functions return the size of arrays. See the example `MatSum` previously.

The lower-bound of arrays created by the **DIM** statement can be changed from the default value of 0 to 1 by the **OPTION BASE** statement, for example:

```
OPTION BASE 1 'arrays now start at one.
```

OPTION BASE is an executable statement, and so its effect depends on the order of execution in the program, not the order of the program text. It is thus possible for an array to have different dimensions if it is **ERASE**d and then **REDIM**med. Using **OPTION BASE** normally only saves a considerable amount of memory if you are using 3 or more dimensions in an array.

When array checks are switched off **OPTION BASE** statements are ignored.

ERASE may be used to free the space used by an array when it is no longer required. This is particularly useful if you have temporary results stored in an array. Once an array has been **ERASE**d you can **DIM** it again.

For example,

```
DIM temp(10000) ' 10000 temporary results
.
.
ERASE temp

' note that temp is not followed by parentheses. This
' anomaly is present for compatibility with other BASICs
```

The **REDIM** statement gives the equivalent of an **ERASE** followed by a **DIM** in one statement. Thus

```
REDIM temp(100)
```

is equivalent to

```
ERASE temp: DIM temp(100)
```

HiSoft BASIC Professional has a powerful extension to let you change the size of arrays whilst retaining their data called **REDIM APPEND.**

For example:

```
SUB AddElement(value)
SHARED table(1), maxentries, nextentry

IF nextentry> maxentries THEN
        ' no room for this entry
        maxentries=maxentries+100
        REDIM APPEND table(maxentries)
        ' the above makes the array 100 elements larger
END IF

table(nextentry)=value  'enter the value
nextentry=nextentry+1   'ready to store the next one

END SUB
```

This example shows how you can avoid fixed limits on the sizes of arrays. If you run out of room just make it a bit bigger. **REDIM APPEND** requires enough memory to make a copy of the array. You can also use **REDIM APPEND** to make arrays smaller; again a copy of the array is made.

Normally the **ERASE**, **REDIM** and **REDIM APPEND** statements cause arrays to be moved in memory. As a result, if there are any pending array elements that have been used in variable parameters, then these will be become invalid. The best way to avoid this is by not passing array elements by reference. For example the following may not work as intended:

```
DIM x(50), a(30)
.
.
Subprog a(3)     ' note variable parameter.
.
SUB Subprog(b)   ' note variable parameter

ERASE x
                 ' a() will now become corrupt, it has
                 ' been moved because it was declared
                 ' after x() which has been erased

.
END SUB
```

Unlike many BASIC compilers, **HiSoft BASIC Professional** will let you change the number of dimensions of arrays with a **REDIM** statement. This may prove useful when porting certain programs that were developed with interpreters, however we recommend strongly that you avoid this as it can make programs almost un-maintainable.

Limitations Imposed by the Compiler

We have tried to avoid placing limits on the programs you can write. For example, most compilers have a limit on the number of characters that are significant in an identifier; **HiSoft BASIC Professional** does not impose any limit on this so that

```
A_very_long_identifier_indeed_which_goes_on_and_on
```

is different from

```
A_very_long_identifier_indeed_which_goes_on_and_on_and_is_d
ifferent
```

This sort of limitation may not seem important to you, but such possible restrictions have the annoying habit of appearing when you think you have nearly finished a large program.

The next section lists the remaining limitations other than the total workspace of the compiler. If you find these restrictive please tell us.

A program may not have more than 12287 lines. If you hit this limit you can probably get round it by having more than one statement per line. We have a 5000 line program which is about 135K bytes of source.

The total number of active labels in the code generation phase of the compiler may not exceed 2999. A label is generated for each line number or label that is referenced (not for those that are un-used) together with 2-3 or each sub-program, 2 for each **CASE** in **SELECT** statements plus 2 for the **SELECT** itself and 2 for most structure statements. For example our 5000 line program requires about 1100 such labels.

There is a limit of approximately 8000 on the number of sub-programs, local variables and parameters in the entire program.

The total number of different names in your program may not exceed 32767.

The total code of a **SELECT** statement may not exceed 32k bytes. To avoid this make some of the alternatives into sub-programs.

The total size of some **FOR...NEXT** loops may not exceed 32k bytes. To avoid this make some or all of the loop into a sub-program.

Sub-programs and user defined functions may not have more than 128 parameters.

The total space for global and **STATIC** local variables and the descriptor table may not exceed approximately 29K bytes. The amount of storage, in bytes, in this area required for the different types is:

2	integers
4	long integers, single precision numbers
8	strings, double-precision numbers, all arrays

The data in strings and arrays is not stored in this area.

Local variable stack space may not be more than 32k bytes per invocation. The different types require the number of bytes given in the table above. There is a limit of 255 channels.

ON...GOTO and **ON...GOSUB** statements may not have more than 8190 line numbers each (!)

Chapter 5
Command Reference

This chapter gives a detailed description of each and every **HiSoft BASIC Professional** statement and function.

It is arranged as follows:

- **Syntax**

This shows the allowable forms of the statement or function. Parameters are denoted in *italics* and optional items are enclosed in square brackets [ ].

- **Effect**

Details a summary of the actions of the statement or function.

- **Comments**

This describes the actions in much greater detail, where required.

- **Example**

Shows one or more examples of the statement or function in use.

ABS function

• Syntax

ABS(*numeric_expression*)

• Effect

This function returns the absolute value of the *numeric_expression*

• Comments

The absolute value function returns the unsigned value of the *numeric_expression*. The absolute values of both -1 and 1 is 1. The type of the result is the same as the type of *numeric_expression*.

• Example

```
PRINT ABS(6*(7)), ABS(6*(-7))
```

Result:

```
42        42
```

AREA statement

• Syntax

```
AREA [STEP](x,y)
```

• Effect

Adds a point to the list to be used by the next **AREAFILL** statement.

• Comments

To draw a filled polygon with **HiSoft BASIC Professional**, specify the points using the **AREA** statement and then draw the polygon with the **AREAFILL** statement.

If the **STEP** keyword is used then the co-ordinates *x* and *y* are treated as relative to the current graphics pen position, otherwise they are treated as relative to the current window.

A maximum of 20 **AREA** statements may be used between calls to **AREAFILL**; subsequent **AREA** statements will be ignored. There is no need to specify the starting point again as the end point.

To draw a polygon that is not filled use the **LINE** statement.

• Example

```
AREA     (50,50)
AREA     STEP (40,0)
AREA     STEP (0,30)
AREAFILL
```

Result:

Draws a filled triangle with corners (50,50), (90,50), (90,80) in the current window.

AREAFILL statement

• Syntax

```
AREAFILL [fill_mode]
```

• Effect

Fills or inverts the polygon specified by AREA statements.

• Comments

fill_mode is an integer expression. If it evaluates to

0 then the area pattern specified by the pattern statement is used;

1 then the area is inverted.

If the *fill_mode* is omitted then the area is filled with the current pattern as if `AREAFILL 0` had been used.

See the **AREA** statement for the details of setting the points.

• Example

```
AREA (20,20)
AREA (30,20)
AREA (60,30)
AREA (20,30)
AREAFILL 1
```

Result:

Inverts a wedge shaped area of the screen.

ASC function

• Syntax

ASC(*string_expression*)

• Effect

This function returns a numeric value that is the ASCII code for the first character of the *string_expression*.

• Comments

If the string passed is a null-string, an `Illegal function call` error is returned. The result is an integer.

• Example

```
X$="FORTY-TWO"
PRINT ASC(X$)           'Prints ASCII code for "F"
```

Result:

```
70
```

ATN function

• Syntax

```
ATN(numeric_expression)
```

• Effect

This function returns the arctangent of the *numeric_expression*, or the angle whose tangent is the *numeric_expression*.

• Comments

The result is returned in radians, in the range of $-\pi/2$ to $\pi/2$ radians.

The *numeric_expression* can be of any numeric type. **ATN** is single precision by default; if the numeric value is double precision, **ATN** returns a double precision value.

• Example

```
PRINT 4*ATN(1);4*ATN(1#)
```

Result:

```
 3.141593       3.141592653589794
```

BEEP statement

• Syntax

```
BEEP
```

• Effect

Flashes the screen.

• Comments

BEEP uses the ASCII bell character; the same effect can be obtained by using `PRINT CHR$(7);`. Unfortunately this does not actually cause any sound to be made. If we had included this ability then every program that printed to the screen would need to include the large amount of code and workspace that producing sound on the Amiga requires.

The equivalent of the interpreter's **BEEP** statement may be achieved using

```
BEEP : SOUND 880,2
```

• Example

```
PRINT "Watch out!"
BEEP
```

BIN$ function

• Syntax

BIN$(*numeric_expression*)

• Effect

This function returns a string which is the binary representation of *numeric_expression*.

• Comments

BIN$ returns the representation of the integer part of *numeric_expression*. If the expression is an integer then the resulting string will be from 1 to 16 characters in length, but if it is a long integer then it can be up to 32 characters long.

• Example

```
PRINT BIN$(8)
```

Result:

```
1000
```

BLOAD statement

• Syntax

```
BLOAD filename,address
```

• Effect

This statement loads a binary file into the buffer specified.

• Comments

The *filename* is a string expression following the standard AmigaDOS conventions. The *address* is a long integer.

The buffer address is the responsibility of the user. **BLOAD** does not check if the address given is a safe address in RAM.

This statement is not provided by AmigaBASIC but is supported by **HiSoft BASIC Professional** on the Atari ST and some versions of Microsoft BASIC.

• Example

```
DIM A%(16000)
BLOAD "PICTURE.DMP",VARPTR(A%(0))
```

This line loads a binary file directly into the array a%(). If the file `PICTURE.DMP` is more than 32000 (16000*2) bytes long then the system may crash as the data will not all be loaded into the array

BREAK statement

• Syntax

```
BREAK {ON|OFF|STOP}
```

• Effect

Modifies the break event trapping to change the effect of **ON...BREAK** statements.

• Comments

`BREAK ON` should be used to enable break event checking. This will cause **ON BREAK** statements to be acted on whenever the user presses `Ctrl-C` or A.

After a `BREAK OFF` statement is these keys will be ignored whilst the program is not waiting for input.

`BREAK STOP` causes menu events to be stored until a `BREAK ON` statement occurs. The **ON BREAK...GOSUB** will then be acted on. This can be useful to suspend menu processing whilst some essential code is executing.

If the break checks compiler option is on then the program will stop when break is pressed, regardless of any **ON BREAK** statements.

• Example

```
ON BREAK GOSUB StopMe
BREAK ON

StopMe:STOP
```

' causes the program to stop if `Ctrl-C` or A. are pressed.

BSAVE statement

• Syntax

```
BSAVE filename,address,length
```

• Effect

This statement saves the contents of the specified buffer to the output device specified in *filename*.

• Comments

It is possible, though not necessarily useful, to **BSAVE** to the parallel or serial ports. Both the address as well as the length of the buffer are long integers. The *filename* must follow the AmigaDOS conventions.

This statement is not provided by AmigaBASIC but is supported by **HiSoft BASIC Professional** on the Atari ST.

• Example

```
BSAVE "ARRAYA",VARPTR(a%(0)),VARPTR((UBOUND(a%)+1)*2)
```

This line saves the entire array a%() to a disk file named ARRAYA.

CALL statement

• Syntax

[CALL] *sub_program_name* [(*parameter* [,*parameter*]…)]

• Effect

Calls a sub-program defined using **SUB...END SUB** or a library routine.

• Comments

sub_program_name is the name of the sub-program being called. There must be the same number of parameters as in the sub-program's declaration.

Further information on sub-programs can be found in **Chapter 4**.

The keyword **CALL** can always be omitted if the sub-program has parameters and if it is not the first statement of a multi-statement line; this is to avoid conflicts with label declarations which are also identifiers followed by a colon. If **CALL** is omitted the brackets round the parameters *must* be omitted.

Array parameters must be passed array names followed by () in **CALL** statements and string parameters must be passed strings. However different numeric types will be converted to the type required by the sub-program.

Parameters are passed by reference only if they consist of just a variable of the same type as in the declaration of the sub-program and the **VAL** keyword was not specified for the parameter when the sub-program was defined. Array elements may be passed as variable parameters.

You can pass a variable by value when it would otherwise be passed by reference simply by enclosing the variable in parentheses.

• Examples

```
QuickSort MyArray(),100
CALL QuickSort(MyArray(),100)   ' these are equivalent
QuickSort(MyArray(),100)        ' syntax error because
                                ' of brackets
call SKIP_ONE:SKIP_ONE  'call required for first call
FRED  X                 ' call by reference
FRED (X)                ' call by value
```

CALL LOC statement

• Syntax

```
CALL LOC address [,parameter]...
```

• Effect

Call a machine-code routine, with the option of passing long integer parameters.

• Comments

This is directly equivalent to the AmigaBASIC machine language **CALL** statement. Unfortunately we had to change the syntax to avoid confusion between machine-code and sub-program **CALL** statements.

The machine-code at *address* is executed, and the parameters can be found on the stack as long integers in C order.

The machine-code can return to the BASIC program by executing a `RTS` statement. All registers except `A7` may be destroyed.

• Example

```
DIM spare%(100)
BLOAD "CALCPI",VARPTR(spare%(0))
calcpi&=VARPTR(spare%(0))
CALL LOC calcpi&,100
```

CALLS statement

• Syntax

CALL *sub_program_variable*

• Effect

Calls a sub-program indirectly using a variable as a pointer for it.

• Comments

sub_program_variable must have been initialised using the **VARPTRS** function to point to a sub-program.

Caution: this is for advanced programmers only. Incorrect use of **CALLS** can wreak havoc on a running program or its memory area.

• Example

```
read_char&=VARPTRS(char_from_disc)
DO
        CALLS read_char&
LOOP UNTIL no_more
.
.
SUB char_from_disc
IF EOF(2) THEN
        read_char&=VARPTRS(char_from_mem)
        CALLS read_char&
        EXIT SUB
ELSE
.
.
.
END SUB
```

CDBL function

• Syntax

CDBL(*numeric_expression*)

• Effect

This function converts the *numeric_expression* to a double precision number.

• Comments

The effect is identical to assigning the *numeric_expression* to a double precision variable and then using the double precision variable.

• Example

```
PRINT ATN(1)              'single precision
PRINT CDBL(ATN(1))        'single extended to double
PRINT ATN(CDBL(1))        'double precision accuracy
```

Result:

```
 .7853982
 .7853981852531433
 .7853981633974485
```

CHAIN statement

• Syntax

```
CHAIN filename
```

• Effect

This statement loads and executes another program.

• Comments

filename should conform to AmigaDOS specifications and be the name of an executable (i.e. double-clickable) program, including any extension. The program does not have to be a **HiSoft BASIC Professional** compiled program.

All files, windows and screens are closed before the program is CHAINed.

• Example

```
DO
        INPUT "File to run";f$
LOOP UNTIL FEXISTS(f$)
CHAIN f$
```

CHDIR statement

• Syntax

CHDIR *pathname*

• Effect

This statement changes the current directory.

• Comments

The *pathname* specified must conform to the AmigaDOS conventions.

It is possible to use relative (as opposed to absolute) path names. If a directory :ONE/TWO exists and it is the current directory, CHDIR "/" will change the current directory to one hierarchic level higher (in this case :ONE).

It is also possible to **CHDIR** one level downwards. To switch from :ONE to :ONE/TWO requires CHDIR "TWO"; CHDIR ":ONE/TWO" is not necessary in this case.

• Example

```
CHDIR ":HBASIC"          'an absolute path
CHDIR "SOURCE"           'this changes the directory
                         'to :HBASIC/SOURCE
```

CHR$ function

• Syntax

CHR$(*ASCII_code*)

• Effect

This function returns a one character string whose *ASCII_code* was passed as the parameter.

• Comments

CHR$ is usually used to produce characters which are not readily available from the keyboard. Common uses are sending a form feed (ASCII 12) to a printer or printing foreign characters on the screen. The *ASCII_code* is an integer.

• Example

```
LPRINT CHR$(12);        'sends a form feed to the printer
PRINT "Copyright ";CHR$(169);" Acme Programming"
```

CINT function

• Syntax

CINT(*numeric_expression*)

• Effect

This function converts the *numeric_expression* to an integer value by rounding its fractional part.

• Comments

If the *numeric_expression* is not in the range -32768 to 32767, an `Overflow error` is returned.

CINT differs from **INT** and **FIX** in that it produces an integer value by rounding. An example of the differences of the three functions can be found under **INT**.

This is equivalent to assigning the *numeric_expression* to an integer variable and then using that variable.

• Example

```
PRINT CINT(1.5),
PRINT CINT(-1.5)
```

Result:

```
 2 -2
```

CIRCLE statement

• Syntax

```
CIRCLE [STEP](x_centre,y_centre),radius
       [,colour_num][,start_angle[,end_angle][,aspect]
```

• Effect

Draws a circle, ellipse or arc in the current window.

• Comments

If the optional angle parameters are omitted, the command draws a hollow circle or ellipse in the current foreground colour (the first parameter in the **COLOR** statement). The angle parameters are expressed in radians between -2π and +2π and indicate where the ellipse or circle is to start and end. Angles are measured anti-clockwise starting on the right side.

x_centre and *y_centre* specify the centre of the circle. If the STEP keyword is used then the circle/ellipse is drawn relative to the current graphics pen position. After the circle has been drawn the current graphics position is set to the centre of the circle.

If *start_angle* or the *end angle* are negative then a line is line drawn connecting the end points to the centre of the circle thus drawing a pie slice. When drawing a pie slice,the angle are treated as if they were positive.

The *colour_num* parameter is used to set the colour of the circle/ellipse and is set using the **PALETTE** statement. If this parameter is omitted then the current foreground pen (as set using the first parameter of the **COLOR** statement is used.

The *aspect* parameter is used to draw ellipses. If *aspect* <1 then the x-radius is taken as *radius* and the y-radius as *radius*aspect*. If *aspect* >1 then the y-radius is taken as y and the x-radius as *radius/aspect*. The default value of *aspect* is 0.44. This is the value that gives a circle on standard Amiga NTSC monitors. If you are using another screen, then you may find that you will need a different value for pure circles.

Solid circles and arcs can be drawn using the **CIRCLE** statement followed by the paint statement.

The *x*, *y*, *radius* and *colornum* parameters are integers. The *start_angle*,*end_angle* and *aspect* parameters are single-precision numbers.

• Examples

```
CIRCLE (200,100),75,,0.' draws a circle 75 pixels wide
                       ' centred on position (200,100)

CIRCLE  (30,30),10,,0,1.57,0.22
CIRCLE  STEP(0,30),10,,0,1.57,0.22
CIRCLE  STEP (30,0),10,,0,1.57,0.22
CIRCLE  STEP (0,-30),10,,0,1.57,0.22
```

CLEAR statement

• Syntax

```
CLEAR
```

• Effect

This statement clears all variables and closes all channels.

• Comments

The following actions are taken by this command:

- All global numeric variables are reset to zero
- All global string variables are reset to a null string (i.e. "")
- All numeric arrays have their contents reset to zero
- All string arrays have their contents reset to a null string
- All files are closed

• Example

```
IF a$="ZERO" THEN CLEAR
```

CLNG function

• Syntax

CLNG (*numeric_expression*)

• Effect

This function converts the *numeric_expression* to a long integer value by rounding its fractional part.

• Comments

If the *numeric_expression* is not in the range -2147483648 to 2147483647, an `Overflow error` is returned.

CLNG is similar to differs from **CINT** but returns a long integer value. It can often be used to ensure that overflow does not take place when assigning the product of integers to a long integer.

This is equivalent to assigning the *numeric_expression* to a long integer variable and then using that variable.

• Example

```
x&=clng(i%)*1000

'is normally better than

x&=i%*1000

'since the latter will give overflow if i% is 33 or more.
```

CLOSE statement

• Syntax

```
CLOSE [[#]channel_number[,[#]channel_number]…]
```

• Effect

This statement terminates I/O to the specified file or device.

• Comments

CLOSE is the opposite of **OPEN**. The *channel_number* is the number specified in the **OPEN** statement for a particular file or device.

CLOSE without parameters closes all **OPEN**ed files and devices.

Once a file is **CLOSE**d, its channel number may be used to **OPEN** any unopened other file or device.

Closing a file or device that was opened for sequential output causes the final buffer to be written before closing.

CLEAR, **END/SYSTEM**, **STOP** and **RUN** close all files and devices automatically.

• Example

```
CLOSE #1,#2
```

CLS statement

• Syntax

```
CLS
```

• Effect

Clears the current output window and returns the cursor to the upper left corner.

• Comments

Only the current window is affected; other windows will be left alone.

COLLISION statement

• Syntax

```
COLLISION {ON|OFF|STOP}
```

• Effect

Modifies the collision event trapping to change the effect of **ON...COLLISION** statements.

• Comments

COLLISION ON should be used to enable collision event checking. This will cause **ON...COLLISION** statements to be acted on whenever an object collides with the border or another object, i.e. the **COLLISION(0)** function would return a non-zero value.

After a **COLLISION OFF** statement is executed, collisions with be ignored.

COLLISION STOP causes menu events to be stored until a **COLLISION ON** statement occurs. The **ON COLLISION...GOSUB** will then be acted on. This can be useful to suspend collision processing whilst some essential code is executing.

• Example

```
COLLISION ON
COLLISION OFF
COLLISION STOP
```

COLLISION function

• Syntax

`COLLISION(object_id)`

• Effect

Provides information about object collisions that have occurred.

• Comments

There are three different flavours of call to the **COLLISION** function:

`COLLISION(-1)` returns the id of the window in which the next collision to be processed occurred.

`COLLISION(0)` returns the object id of the object that was involved in the next collision to be processed.

`COLLISION(n)` where `n>0` is used to find out the type of collision that involved object id n; this also removes this collision from the list of pending events. n is normally found by using `COLLISION(0)`. The value returned from `COLLISION(n)` is one of:

-1	Top Border
-2	Left Border
-3	Bottom Border
-4	Right Border
>0	Object id of the object with which object n collides.

HiSoft BASIC Professional maintains a queue of up to 16 events in the same way as AmigaBASIC.

• Example

```
obj%=COLLISION(0)
IF obj%>0 THEN
        sort%=COLLISION(obj%)
        SELECT CASE sort%
        CASE -1:        'Top Border code here
        CASE -2:        'left Border code here
        CASE -3:        'Bottom Border code here
        CASE -4:        'Right Border code here
        CASE ELSE       'hit another object
        END SELECT
END IF
```

COLOR statement

• Syntax

```
COLOR [foreground_col][,background_col][,drawing_mode]
```

• Effect

Sets the colours to be used in the current window.

• Comments

This sets the *foreground_colour* for the current window which is used for drawing points, lines, text, and for filling areas. The *background_colour* is used for the screen background (e.g. when using **CLS**).

The numbers used to identify colours can be set by using the **PALETTE** statement and the **Preferences** tool.

The default values for the colours in the Workbench screen are:

0 blue
1 white
2 black
3 orange

If one colour parameter is omitted then this attribute remains the same.

The optional *drawing_mode* parameter is a HiSoft extension which sets the drawing mode without the need to call the Graphics library directly. Valid drawing modes are:

0 **JAM1**
the foreground colour is "jammed" into where the graphics are drawn. The background colour is ignored.

1 **JAM2**
the current pattern is used with the foreground colour is used for ones in the pattern and the background colour for zeros. This is the default.

2 **COMPLEMENT**
for each one bit that is drawn the colour is complemented; e.g. when using 4 colours: colour 0 becomes colour 3, colour 1 becomes colour 2 and vice versa.

4 **INVERSEVID** and **JAM1**
causes the pixels surrounding text to be drawn in the foreground colour and the actual text pixels to be left as they were.

5 **INVERSEVID** and **JAM2**
causes text to be drawn in the background colour surrounded by the foreground colour thus giving the same affect as "inverse video" on old fashioned computers.

• Examples

```
COLOR 2,3
CLS
PRINT "fills the window with black this is in orange"
COLOR ,,5
PRINT "This is inverse video"
COLOR ,,2
PRINT "Back to normal"
```

COMMAND$ function

• Syntax

```
COMMAND$
```

• Effect

This function returns the command line of the program.

• Comments

This function enables a program to access the command line entered if the program is run from the CLI.

Any leading spaces are removed from the command line. However, unlike some other machines the command line is *not* upper-cased.

If a program is run from the workbench then the function returns a null string, though this may change in a future release.

• Example

```
OPEN COMMAND$ AS #1      ' a filename passed on the command
                         ' line is used to open a file
```

COMMON SHARED statement

• Syntax

```
COMMON SHARED variable [ (subscripts…)][,variable [
        (subscripts…)]]…
```

• Effect

This statement indicates that global variables may be accessed throughout any sub-programs .

• Comments

This statement is provided mainly for QuickBASIC compatibility.

It has the same effect as the **DIM SHARED** statement except that the subscripts of arrays are ignored. Arrays used in this statement should be **DIM**med before the **COMMON SHARED** statement.

• Example

```
DIM glob(20)
COMMAND SHARED debug_flag,glob(1)
```

is equivalent to

```
DIM SHARED debug_flag,glob(20)

' both debug_flag and glob() can now be accessed anywhere
in the 'program.
```

CONST statement

• Syntax

CONST *name=integer_constant* [,*name=integer_constant*]...

• Effect

CONST defines symbolic constant values for use in place of integer variables.

• Comments

Constants must be declared before they are used and their names must be valid integer variable names. *integer_constant* must be a simple integer number or character constant optionally preceded by a minus sign.

Constants can be used to make code that accesses data structures in memory much easier to read and also to give a loose equivalent to user enumerated types in Pascal.

Constants may be used within sub-programs and user-defined functions without declaring them as **SHARED**.

• Example

```
DEFINT a-z
CONST white=0,black=1,red=2,green=3
COLOR red,green,black
CONST firstletter="A"%,lastletter="Z"%
IF (c%>=firstletter) AND (c%<=lastletter) THEN
        PRINT "It's a lower case letter"
```

COS function

• Syntax

COS(*numeric_expression*)

• Effect

This function returns the cosine of the *numeric_expression* , which must be in radians.

• Comments

COS is normally performed in single precision. If the *numeric_expression* is double precision, **COS** is performed in double precision.

• Example

```
PRINT COS(0)
```

Result:

```
1
```

CSNG function

• Syntax

CSNG(*numeric_expression*)

• Effect

This function converts *numeric_expression* to a single precision number.

• Comments

CSNG has the same effect as assigning *numeric_expression* to a single precision variable and then using that variable.

• Example

```
PRINT SIN(1#)
PRINT CSNG(SIN(1#))
```

Result:

```
.8414709848078965
.841471
```

CSRLIN function

• Syntax

```
CSRLIN
```

• Effect

This function returns the current line position of the cursor in the current window.

• Comments

To return the current column position, use **POS**. The value returned is an integer, with the top of the screen (or window) returning the value of 1, corresponding to the **LOCATE** statement. If the current output window has been closed then the value 1 is returned.

• Example

```
L%=CSRLIN
C%=POS(0)
PRINT "The cursor was on line";L%;"and in column";C%
```

Result (if the cursor is on line 2, column 3):

```
The cursor was on line 2 and in column 3
```

CVD,CVFFP,CVI,CVL,CVS functions

• Syntax

CVD(*8-byte string of a double precision float*)

CVFFP(*4-byte string of a single precision float*)

CVI(*2-byte string of an integer*)

CVL(*4-byte string of a long integer*)

CVS(*4-byte string of a single precision float*)

• Effect

These functions return the internal numeric values of strings of bytes.

• Comments

These functions are the counterparts to **MKD$**, **MKFFP$**,**MKI$**, **MKL$** and **MKS$**. They do not change the value of the actual data, they only induce BASIC to interpret them differently.

The most common use for these functions is for processing random access files in which numeric values have been stored as strings by the **MKD$**, **MKFFP$**, **MKI$**, **MKL$** or **MKS$** functions.

The **CVFFP** function is provided as a more efficient way of storing single-precision numbers than **CVS** but is *not* compatible with files produced or readable by the interpreter.

• Example

```
D#=9/11
I%=10
L&=42
S!=5/7
PRINT D#,I%,L&,S!
OPEN "TEST.DAT" FOR OUTPUT AS #1
PRINT #1,MKD$(D#);MKI%(I%);MKL(L&);MKS(S!)
CLOSE #1
'The converted values of the different variables are
'binary. Printing these values to screen would result in
'strange looking control characters being displayed.
OPEN "TEST.DAT" FOR INPUT AS #1
D1$=INPUT$(8,#1)
I1$=INPUT$(2,#1)
L1$=INPUT$(4,#1)
S1$=INPUT$(4,#1)
CLOSE #1
PRINT CVD(D1$),CVI(I1$),CVL(L1$),CVS(S1$)
```

Result:

```
 .8181818127632141      10      42      7142857
 .8181818127632141      10      42      7142857
```

DATA statement

• Syntax

DATA *constant* [,*next_constant*]…

• Effect

This statement defines the data to be used by the **READ** statement.

• Comments

DATA is not executed. It may contain as many constants as will fit on a line. **READ** will read **DATA** in the order of their appearance in the program. Multiple **DATA** statements will be processed as one continuous string of constants.

Constants may be of any variable type. If necessary the data will be converted into the correct type for the variable being **READ**.

It is not necessary for quotation marks to be present around string constants in **DATA** unless the string contains colons, commas or leading and trailing spaces. Numeric values may be preceded by minus signs.

A null item in the list of constants is allowed. When read, a numeric variable it is assigned the value 0. If read as a string variable, the value will have the null value.

READ can process **DATA** statements again if **RESTORE** is used. Otherwise it is not possible to re-read **DATA**.

• Example

```
DATA 2.2,1.6,"Hello"
READ a,b%,c$
PRINT a;b%;c$
```

Result:

```
 2.2  2 Hello
```

The data item 1.6 is rounded before being assigned to the integer b%, hence the result of 2.

DATE$ function

• Syntax

```
DATE$
```

• Effect

This function returns the current date.

• Comments

The returned value is a ten character string. The format is *mm-dd-yyyy*.

• Example

```
PRINT "Today's date is: ";DATE$
```

DECLARE statement

• Syntax

```
DECLARE {SUB|FUNCTION} subname [(param_list)] [LIBRARY]
```

• Effect

Declares the parameters of a sub-program or function and indicates that a library function is a function rather than a sub-program.

• Comments

When the **LIBRARY** keyword is used the parameter list is ignored; When used for non-**LIBRARY** procedures the parameter list is in the same form as with **FUNCTION** and **SUB** definitions. See **Chapter 4** for more details.

• Example

```
LIBRARY "graphics.library"
DECLARE FUNCTION SetSoftStyle& LIBRARY

DECLARE FUNCTION Factorial!(n%)
...
PRINT Factorial!(4)
...
FUNCTION Factorial!(n%)
...
END FUNCTION
```

DECR statement

• Syntax

```
DECR numeric_variable
```

• Effect

Subtracts one from the specified variable

• Comments

numeric_variable may be a simple numeric variable or a numeric array element of any type. If you wish to decrement a simple variable `X` it probably makes more sense stylistically to use `X=X-1` rather than `DECR X`, but if you have a complicated array expression as below `DECR` is the appropriate construct.

• Example

```
DECR A(B(I,j+5)) 'equivalent to A(B(I,j+5))=A(B(I,j+5))-1
```

DEF FN statement

• Syntax

```
DEF [FN]function_name[(parameter_list)]=expression

                    or

DEF [FN]function_name[(parameter_list)]
[LOCAL variable_list]
[STATIC variable_list]
[SHARED variable_list]
.
.
statements
.
.
[EXIT DEF]
[FNfunction_name=expression]
.
.
END DEF
```

• Effect

Define single- or multi-line functions with parameters.

• Comments

The *function_name* must be a unique identifier in your program. The optional *parameter_list* can contain any number of variable names separated by commas, each of which may be preceded by **VARPTR** to denote variable rather than value parameters (see **Chapter 3** and **Chapter 4** for more details of parameter types).

The *function_name* must be a variable name. For compatibility with other BASICs the name should start with **FN** .The type returned is determined by the same rules as for variables except that if the name starts with **FN**, the **FN** is ignored. Thus `FNfred%` returns an integer. `FNjohn` will normally return a single precision floating point value unless you have used a **DEF*type*** statement affecting the letter `J`. Note that `john` and `FNjohn` are different names.

Inside **DEF** function definitions, variables are assumed to be global unless you use a **STATIC** or **LOCAL** statement to override this.

Function definitions may be recursive. Only functions beginning with **FN** may be called before they are declared. However function definitions can not be nested inside other function or sub-program definitions or within structured control statements.

[FN]*function_name=expression* is used to return values from multi-line functions. You can have more than one such statement within a function definition.

EXIT DEF causes the current function to be exited immediately.

If you call multi-line functions from within expressions, be careful that you do not have problems with side effects. For example

```
DEF FNx
  j=j+1
  FNx=j
END DEF
j=4
PRINT FNx+j
```

will probably not produce the same result as

```
j=4
PRINT j+FNx
```

because the variable `j` is modified when the function is called.

It is also dangerous to use input/output statements within multi-line functions if you then call them from within another input/output statement. For example,

```
PRINT 54,FNsilly,j
DEF FNsilly
PRINT " Function silly is being called."
FNsilly=1.4
END DEF
```

is not sensible.

The best idea is either never to call functions in input/output *or* never to input/output within functions.

- **Example**

```
DEF FNahex$(a&)="&H"+hex$(a&)
DEF FNlonghex$(a&)
LOCAL a$
a$=hex$(a&)
IF LEN(a$)<8 THEN
        a$=STRING$(8-LEN(a$),"0")+a$
END IF
FNLonghex$=a$
END DEF
PRINT FNahex$(42),FNlonghex$(42)
```

DEFDBL, DEFINT, DEFLNG, DEFSNG, DEFSTR statements

• Syntax

```
DEFDBL letter_range [,letter_range]...

DEFINT letter_range [,letter_range]...

DEFLNG letter_range [,letter_range]...

DEFSNG letter_range [,letter_range]...

DEFSTR letter_range [,letter_range]...
```

• Effect

These statements declare variables to be double precision, integer, long integer, single precision, or string.

• Comments

The format of the *letter_range* is *1st_letter [-2nd_letter]*. Allowable letters are upper- or lowercase. Any variable name beginning with a letter specified in a **DEF*type*** statement defines the value of the variable: **DBL** for double precision, **INT** for integer, **LNG** for long integer, **SNG** for single precision, or **STR** for a string.

The type declaration characters **#,%,&,!,$** override any **DEF*type*** statements.

Note The variables `I#`, `I%`, `I&`, `I!`, and `I$` are all different variables with potentially different values.

• Example

```
DEFINT a-z      ' a good idea to have at the front of all
                ' programs so you only use floating point
                ' when needed and avoids many %s in names.
DEFINT i-n
DEFSNG a-h,o-z  ' for old-fashioned FORTRAN programmers
                ' but confusing for everyone else.
```

DIM statement

• Syntax

```
DIM [SHARED]variable [ (subscripts…)]
        [,variable [ (subscripts…)]]…
```

• Effect

This statement defines the maximum values for array-variable subscripts and allocates the necessary storage.

• Comments

The maximum subscript value for an unDIMmed array is 10. The maximum number of dimensions in an array is 31 (which would take up a minimum of 4 GigaBytes of memory if each index had more than one possible element). Subscript values can be integer or long integer expressions.

It is good programming practice to dimension all arrays even if you require 10 elements.

The minimum value for each subscript may be set using the **OPTION BASE** statement.

All arrays are dynamic and can be modified with **ERASE**, **REDIM**, and **REDIM APPEND**.

When initially defined, the elements of a numeric array have the value 0. If a string array is defined, the elements have null values.

The **DIM SHARED** form is used to declare arrays and ordinary variables that can be accessed from within sub-programs without the need for a **SHARED** statement in each sub-program.

- **Example**

```
DIM R(15)
FOR I%=1 TO 15
        R(I%)=I%
        PRINT R(I%);
NEXT I%
```

Result:

```
 1   2   3   4   5   6   7   8   9   10   11   12   13   14   15
DIM SHARED debug_flag   'debug_flag can now be
                        'accessed anywhere in the program.
```

DO...LOOP statement

• Syntax

```
DO [{WHILE|UNTIL} boolean_expression]
.
.
[EXIT {LOOP|DO}]
.
.
{WEND|LOOP}[{WHILE|UNTIL} boolean_expression]
```

• Effect

Repeats the statements within the **DO...LOOP** while the conditions are true (**WHILE**) or false (**UNTIL**).

• Comments

The *boolean_expression*s must be numeric expressions. If they evaluate to 0 this is taken as false; non-zero as true.

If an **EXIT LOOP** or **EXIT DO** statement is contained within the loop then control passes to the statement following the end of the loop. If **DO...LOOP**s are nested the inner most loop is exited. If you wish to exit an outer loop use the **REPEAT** loop construct. There may be a condition at both ends of the loop if desired or none at all; this is thus considerably more flexible than the **WHILE..WEND** loop.

• Examples

```
i=0
DO
        i=i+1
        PRINT i;
LOOP UNTIL i>=10
```

Or

```
i=0
DO
        i:=i+1
        IF i>10 THEN EXIT LOOP
    PRINT i
LOOP
```

```
Result of both examples:  1  2  3  4  5  6  7  8  9  10
```

END statement

• Syntax

```
END

END DEF

END FUNCTION

END IF

END REPEAT name

END SELECT

END SUB
```

• Effect

Ends a BASIC program, function definition, **IF...THEN...ELSE** block, **SELECT** statement, **REPEAT** loop or sub-program.

• Comments

The **END DEF** statement indicates the end of a **DEF FN**.

END FUNCTION completes a **FUNCTION** definition.

END IF finishes an **IF...THEN...ELSE** block.

END REPEAT ***name*** ends a **REPEAT** loop.

END SELECT ends a **SELECT CASE** statement.

END SUB ends a BASIC sub-program; it is the counterpart of **SUB**. See the relevant statement for more details.

END alone terminates the entire BASIC program. The **END** statement may be put anywhere in the source.

- **Example**

```
DEF FNfactorial&(N%)
        IF N% <= 1 THEN
                FNfactorial&=1
        ELSE
                FNfactorial& = FNfactorial&(N%-1)*N%
        END IF
END DEF
```

EOF function

• Syntax

EOF(*channel_number*)

• Effect

This function tests for the end-of-file condition of *channel_number*.

• Comments

-1 is returned if true; 0 if false. A common use for **EOF** is to test for the end-of-file when writing data to a sequential file. When writing to a random-access file, **EOF** returns true if the last **GET** was not able to read an entire record, due to an attempt to read beyond the end-of-file. The result returned is an integer.

• Example

```
OPEN "TEST.DAT" FOR INPUT AS #1
Ctr=0
WHILE NOT EOF(1)
        junk$=INPUT$(1,#1)
        Ctr=Ctr+1
WEND
CLOSE #1
PRINT Ctr                'The length of the file in
                         'bytes can use LOF instead.
```

ERASE statement

• Syntax

ERASE *array_name* [, *array_name*]...

• Effect

This statement de-allocates arrays previously defined with **DIM**.

• Comments

An array once **ERASE**d can only be accessed again after is has been **DIM**med. **REDIM** alone will **ERASE** and **DIM** an array together, whereas **REDIM APPEND** allows an array to be extended or truncated without losing its remaining data. This command can move existing arrays in memory, so extreme care should be taken if used within functions or sub-programs that are called with arrays or array elements as variable parameters. See **Chapter 4** for more details.

• Example

```
DIM Arr(15000)

' some processing with Arr()

ERASE Arr                ' free the space used by Arr
```

ERL, ERR functions

• Syntax

```
ERL

ERR
```

• Effect

These functions return error status of the current program.

• Comments

When running an error handling routine, **ERR** will return the error code and **ERL** returns the line number in which the error occurred.

Note If the error occurred on a line without a line number, the most recent line number found in the program will be returned in **ERL**.

• Examples

```
IF ERR=53 THEN
        PRINT "File not found error on or after line ";ERL
END IF
.

99 PRINT "FINISHED"
   RETURN
        I=0
        PRINT 1/I
'After executing 1/I the value of ERL will be 99 even
'though this line number has nothing to do with the code
'executed.
```

ERROR statement

• Syntax

ERROR *integer_expression*

• Effect

This statement simulates the occurrence of a BASIC run time error.

• Comments

ERROR is the counterpart to **ERR** in that *integer_expression* is the code returned by **ERR**. The user can also create his own error codes by using error numbers not already used by BASIC.

• Example

```
IF Tst <= 0 THEN ERROR 99       'An error code not reserved
                                'by BASIC
IF ERR = 99 THEN
        PRINT "Something has gone terribly wrong in SUB
TEST"
END IF
```

EXIT statement

• Syntax

EXIT {DEF|DO|IF|FOR|FUNCTION|LOOP|SELECT|SUB|*identifier*}

• Effect

Exits a function definition, BASIC sub-program or a structured statement.

• Comments

EXIT DEF exits from a **DEF** function definition, whereas **EXIT FUNCTION** exits from a function defined with the word **FUNCTION**. **EXIT SUB** exits from a sub-program. **EXIT DO**, **EXIT IF**, **EXIT FOR**, **EXIT LOOP** and **EXIT SELECT** cause the corresponding structured statement to be exited immediately and execution continues with the statement after the corresponding **END**.

EXIT *identifier* causes the named **REPEAT** loop to finish prematurely.

These statements are used to leave routines (e.g. as part of an **IF...THEN...END IF** block). They do not define the end of the routines definitions; **END DEF**, **END SUB** and **END REPEAT *identifier*** are required to do this.

• Example

```
SUB Fred
.
.
IF Var&=12654 THEN EXIT SUB
.
.
END SUB
FOR i=1 TO max
        IF a(i)=0 THEN EXIT FOR
NEXT i
IF i>max THEN
        PRINT "no zero elements"
ELSE
        PRINT "first zero element is";i
END IF
```

EXP function

• Syntax

EXP(*numeric_expression*)

• Effect

Calculates the exponential function of the *numeric_expression*.

• Comments

This function returns *e* to the power of the *numeric_expression*. *e* is the base of natural logarithms. If the *numeric_expression* is double precision then the result is calculated to double precision otherwise single precision is used.

• Example

```
PRINT EXP(1#)
```

Result:

```
2.718281828459046
```

FEXISTS function

• Syntax

FEXISTS(*filename*)

• Effect

Determines whether or not a particular file exists.

• Comments

filename should be a legal AmigaDOS filename, together with a device specifier and sub-directory if required. The function returns -1 if the file is found, or 0 if it is not.

• Example

```
f$="TEST.DAT"
IF NOT FEXISTS(f$) THEN
        f$="B:\"+f$
        IF NOT FEXISTS(f$) THEN
                do_error "Cannot find data"
        END IF
END IF
OPEN f$ FOR INPUT AS #1
```

FIELD statement

• Syntax

```
FIELD [#]channel_number, field_width AS string_variable...
```

• Effect

This statement allocates space for variables in a random access buffer.

• Comments

The *channel_number* is the number which was used to **OPEN** the file. *field_width* is the amount of space in the *string_variable* to be allocated, in characters.

The total number of bytes allocated must not exceed the record length specified when opening the file; default record length is 128 bytes.

When a file is **CLOSE**d, all **FIELD** definitions are lost. All fielded strings that BASIC associates with the file receive null values.

A **FIELD** statement does not cause data to be put into the buffer. It associates a specified amount of space within the random-access buffer with a variable.

Note Do not use a fielded variable in an **INPUT** statement if it is to remain fielded. An **INPUT** or other assignment will cause the variable's pointer to refer to string space, not the random-access buffer.

• Example

```
'The first definition allocates the 12 byte buffer as the
'country code, area code and phone number; the second
'allocates the same buffer entirely to the single
'variable of Wholenum$.
OPEN "JOHN.DAT" FOR INPUT AS #1 LEN=12
FIELD #1, 2 AS Ccode$, 3 AS Acode$, 7 AS Number$
FIELD #1, 12 AS Wholenum$
```

FILES statement

• Syntax

```
FILES [file_spec]
```

• Effect

This statement lists the names of files in the specified directory on a disk.

• Comments

The *file_spec* must be a string conforming to AmigaDOS conventions and may contain a device a number and/or a pathname. and optionally a filename. Please note that for compatibility with the interpreter and to keep the run-time routine compact, wildcards are *not* supported.

• Example

```
FILES "DF0:DEVPAC"      'This produces a list of all
                        'the files in the DEVPAC
                        'directory of the disk in
                        'drive DF0:
```

FILL statement

• Syntax

```
FILL x_start,y_start
```

• Effect

This statement fills enclosed shapes already drawn on the screen..

• Comments

The pixel in the current output window specified by x_start and y_start is plotted and the pattern `spreads out' to fill the whole of the shape. The current fill color, style, and index, as selected by the COLOR statement aare used.

• Example

```
COLOR 1,1,1,0,0,
BAR 70,40,160,120                'hollow box
ELLIPSE 150,100,70,50            'and ellipse
COLOR 1,1,1,1,4
```

Fill 71,50

FIX function

• Syntax

FIX(*numeric_expression*)

• Effect

This function produces the truncated integer part of the *numeric_expression*.

• Comments

The difference between **FIX** and **INT** is that if the *numeric_expression* is negative, **FIX** will return the first negative integer greater than itself, whereas **INT** returns the first negative integer less than the *numeric_expression*. The type of the result will be the same as the type of *numeric_expression*.

• Example

```
PRINT FIX(1.5)
PRINT FIX(-1.5)
```

Result:

```
 1
-1
```

FOR...NEXT statement

• Syntax

```
FOR counter = start TO end [STEP increment]

.

.

NEXT [counter ][,counter]…
```

• Effect

This statement executes a series of instructions in a loop a specified number of times.

• Comments

counter	is a variable used as the loop counter
start	is the value of *counter* at the beginning of the loop
end	is the value of *counter* upon which the loop ceases to execute
increment	is the value by which *counter* is incremented each time the loop is executed

All four values may be either integers, long integers, single or double precision floats.

The program lines between **FOR** and **NEXT** are executed repeatedly. *counter* is adjusted by *increment* after execution of each iteration of the loop. *counter* is then compared with *end*. If *counter* is greater than *end*, execution of the program continues on the line after **NEXT**.

Note that *counter* is updated before the comparison is made.

Unless **STEP *increment*** is specified, the increment value is one. If *increment* is negative, *end* must be less than *start* otherwise the loop will not be executed.

If *start* is less than *end* and *increment* is positive then the loop isn't executed.

FOR...NEXT loops may be nested, provided that each loop has a unique *counter* variable. The **NEXT** statement for the inside loop must appear before the **NEXT** statement for the outside loop.

A **NEXT** statement of the format `NEXT x,y,z` is the same as

```
                NEXT x
        NEXT y
NEXT z
```

A **NEXT** statement without specifying *counter* will match the most recent **FOR** statement.

FOR...NEXT loops may finish prematurely via the use of **EXIT** or **GOTO** statements.

Note The sum of *end* and *increment* (even of *increment* isn't explicitly specified, and is therefore one) must never exceed the range of *counter*'s variable type otherwise overflow results.

Optimisation: A `FOR...NEXT` loop executes fastest when *counter* is a integer and *start*, *end* and *increment* are constants. If any of the last three values must be variables, they should be made integers for maximum speed.

• Example

```
DEFINT I
FOR I_ONE = 1 TO 5
        FOR I_TWO = 3 TO 11 STEP 2
                FOR I_THREE = 0 TO -15 STEP -1
.
.
NEXT I_THREE,I_TWO,I_ONE
```

FRE function

• Syntax

FRE(*numeric_expression*)

FRE(*string_expression*)

• Effect

This function returns the size of free heap space or AmigaDOS free memory.

• Comments

FRE returns the following values:

Different values of *numeric_expression* give different results as follows:

-1 the number of total free bytes in the system .
-2 the number of bytes of stack space that are not used.
-3 gives the largest free block of memory
any other value
free space on the BASIC heap

For a *string_expression*, the size, in bytes, of contiguous free heap space after a garbage collect is returned. The *heap* is the area of memory used by a BASIC program for storing string variables and arrays.

• Example

```
PRINT FRE(0)
PRINT FRE("")          'This is a null string
PRINT FRE(-1)
```

Result:

```
 127534
 154293
 4096
```

GET file I/O statement

• Syntax

```
GET [#]channel_number[,record_number]
```

• Effect

This statement reads a record from a random-access disk file into a random access file buffer.

• Comments

The *channel_number* is the number under which the file was opened. If *record_number* is omitted, the next record after the last **GET** is read.

EOF is useful to use in conjunction with **GET** to check if **GET** was beyond the actual end of the file.

• Example

```
'This routine prints the names and ages of the first 10
'people listed in AGES.DAT
DEFINT I
OPEN "AGES.DAT" FOR RANDOM AS #1
FIELD #1, 10 AS Fname$, 10 AS Lname$, 2 AS Age$
FOR I = 1 TO 10
        GET #1,I                'Read the first 10 entries
in AGES
        PRINT Fname$;" ";Lname$;" is ";Age$;" years old."
NEXT I
CLOSE #1
```

GET graphics statement

• Syntax

`GET(x1,y1)-(x2,y2),array_name[(index_expression...)]`

• Effect

This statement stores binary images from any part of the current output window.

• Comments

GET transfers a rectangular image defined by `(x1,y1)-(x2,y2)` into the array specified by `array_name`. **PUT** is the counterpart statement; it transfers the image from the array to the screen.

`(x1,y1)-(x2,y2)` represent diagonally opposite corners of the rectangle on the screen.

The size (in bytes) of an array needed to store an image is

```
2*p*(y2-y1+1)*[(x2-x1)\16+1] + 6
```

where `p` is the number of planes of screen memory, 2 is the default.. For example to store an image of (10,20)-(100,200) in normal resolution would require 2*2*181*6+6=4350 bytes, or an integer array of 2175 elements.

If an integer array `p%()` is used in a **GET** statement then

`p%(0)`	contains the width of the image
`p%(1)`	contains the height of the image
`p%(2)`	contains the number of planes of the image.

The form of array used by **HiSoft BASIC Professional** is the same as that used by AmigaBASIC. However this is different to **HiSoft BASIC Professional** on the Atari ST.

Note that a multi-dimensional array may be used, making it easy to switch between different images.

- **Example**

```
DIM p%(2174)
AREA 30,30: AREA 30,60: AREA 60,30: AREAFILL
GET (10,20)-(100,200),p%
PUT (30,40),p%
```

GOSUB...RETURN statements

• Syntax

```
GOSUB {line_number1|line_label1}

.

.

RETURN {line_number2|line_label2}
```

• Effect

These statements branch to, and return from, a subroutine, respectively.

• Comments

line_number1 or *line_label1* identifies the first line of the subroutine to which GOSUB branches. You can also **RETURN** from a subroutine to *line_number2* or *line_label2* and not necessarily to the line after the **GOSUB** call. This is not generally recommended as it can lead to very unreadable code.

A subroutine may contain more than one **RETURN** statement. A **RETURN** without *line_number2* or *line_label2* branches back to the line after the **GOSUB** that called the subroutine.

Note We suggest that you use **CALL**s to **SUB...END** subprograms rather than **GOSUB...RETURN**. **SUB...END** routines are much more flexible in that parameters can be passed to them and they can support local variables.

HiSoft BASIC Professional does not support **GOSUB** to a subprogram.

• Example

```
I% = 0
PRINT "Main Loop"
GOSUB Loop1
Back_In_Main:
PRINT "Back in Main Loop"
I% = 1
GOSUB Loop2
PRINT "Done.
END
.
.
Loop1:
PRINT "In 1st subroutine"
RETURN Loop2
Loop2:
PRINT "In 2nd subroutine"
        IF I% = 0 THEN
                RETURN Back_In_Main
        ELSE
                RETURN
        END IF
```

Result:

```
Main Loop
In 1st subroutine
In 2nd subroutine
Back in Main Loop
In 2nd subroutine
Done
```

GOTO statement

• Syntax

```
GOTO {line_number|line_label}
```

• Effect

This statement causes program execution to unconditionally jump to the line specified.

• Comments

GOTOs must *not* be used to enter or leave functions, subprograms or subroutines. **GOTO**s can be used within these structures if necessary.

Note We suggest the use of structured control statements such as **IF...THEN...ELSE** and **DO...LOOP** for enhanced readability of the source and easier debugging.

• Example

```
GOTO Label
'The program never gets here
.
.
Label:
PRINT "The program resumes execution here"
```

HEX$ function

• Syntax

HEX$(*numeric_expression*)

• Effect

This function returns a string that represents the hexadecimal value of the *numeric_expression*.

• Comments

The *numeric_expression* is rounded before being evaluated by **HEX$**. If the parameter is an integer then the resulting string will be from 1 to 4 characters in length, and if it is a long integer then the result may be up to 8 characters in length.

• Example

```
'This routine prints the numbers 1 to 15 in hexadecimal
'notation
DEFINT I
FOR I=1 TO 15
        PRINT HEX$(I);
NEXT I
```

Result:

```
123456789ABCDEF
```

IF...THEN...ELSE statement

• Syntax

```
IF boolean_expression THEN
        statement_1
        [statement_2]
.
.
[ELSEIF boolean_expression THEN
        statement_3
        [statement_4]]
.
.
[ELSE
        statement_5
        [statement_6]]
.
.
END IF
```

OR

```
IF boolean_expression THEN statement_1 [ELSE statement_2]
```

• Effect

This statement block allows conditional execution or branching, based on the evaluation of a Boolean expression.

• Comments

In the single-line variation of **IF...THEN...ELSE**, *statement_1* is executed if *boolean_expression* evaluates to true. If *boolean_expression* is false, then *statement_2* is executed. If **ELSE** is not present and *boolean_expression* is false, the program resumes execution on the line after the **IF...THEN** statement.

- **Example**

```
S!=RND
IF S!<0.5 THEN
        PRINT "S! is smaller than 0.5"
ELSEIF S!>0.5 THEN
        PRINT "S! is larger than 0.5"
ELSE
        PRINT "S! is equal to 0.5"
END IF
```

INCR statement

• Syntax

```
INCR numeric-variable
```

• Effect

This statement adds one to the *numeric_variable*.

• Comments

The *numeric_variable* may be a simple or array variable of any numeric type, the value of which is incremented by one.

• Example

```
INCR a(i,j)      'equivalent to a(i,j)=a(i,j)+1
```

INKEY$ function

• Syntax

```
INKEY$
```

• Effect

This function reads a character from the keyboard without echo, if one is available.

• Comments

`INKEY$` returns either a null, or one byte string containing the character read from the keyboard.

A null string result (i.e. `""`) means no key was pressed.

A one-character string means a normal key was pressed, and contains the ASCII value of it. The following keys return special values:

F1	128	Backspace	8
F2	129	Return	13
F3	130	↑	28
F4	131	↓	29
F5	132	→	30
F6	133	←	31
F7	134	Del	127
F8	135	Help	139
F9	136		
F10	137		

Note that the Amiga keys have no effect on the values returned by `INKEY$`.

• Example

```
'This block provides a quicker way of processing user
'input, if only one character is needed.
PRINT "Enter your choice (1 or 2)"
DO
        a$=INKEY$
        SELECT CASE a$
                CASE ="1"
                        CALL One: EXIT LOOP
                CASE ="2"
                        CALL Two: EXIT LOOP
        END SELECT
LOOP
```

INPUT statement

• Syntax

```
INPUT [;]["prompt"{;|,}] variable_list
```

• Effect

This statement prompts the user for input which is assigned to *variable_list*.

• Comments

INPUT causes program execution to halt and await user data. If a *prompt* is included, **INPUT** will first print it and then a question mark. If the question mark is to be suppressed, a comma should be put after *prompt*. If the cursor is to stay on the same line even after user presses enter (i.e. not echo the CR-LF), a semi-colon should be put immediately after the **INPUT** statement.

• Example

```
INPUT "The square of",x%
PRINT "is";(x%*x%)
```

Result:

```
The square of 2 is 4
```

INPUT# statement

• Syntax

```
INPUT #channel_number,variable_list
```

• Effect

This statement reads data from the device or file specified by *channel_number* and assigns it to *variable_list*.

• Comments

The *channel_number* is the number specified when **OPEN**ing the file.

The data read must be numeric for numeric variables and leading carriage returns, line feeds, and spaces are ignored; the first character that is not one of the three preceding types is considered to be the beginning of a number. A number is terminated by space, carriage-return or a line feeds. If end-of-file is reached while a value is being read, the variable is terminated.

• Example

```
INPUT #1, Str$, Int%, Lng_int&, Sing!, Dub_float#
```

INPUT$ statement

• Syntax

```
INPUT$(n[,[#]channel_number])
```

• Effect

This statement reads *n* characters from the specified channel.

• Comments

If *channel_number* is omitted, the characters are read from the keyboard without echo. This can be used for input similar to **INKEY$**, but with a fixed amount of characters to be processed from the keyboard. *n* may be an integer or long integer. Thus assuming there is sufficient memory available an entire file can be read with one use of this function.

• Example

```
'This program reads the first 512 bytes of a file into the
' string variable first$
OPEN "a.data" FOR INPUT AS #1
first$=INPUT$(512,#1))
CLOSE #1
```

INSTR function

• Syntax

INSTR([*start*,]*1st_string*,*2nd_string*)

• Effect

This function returns the location of the first occurrence of *2nd_string* in *1st_string*.

• Comments

start is an optional offset for the beginning of the search within *1st_string*. *start* may be either an integer or a long integer.The strings may be either string-variables, string-literals, or string expressions.

INSTR will return a value depending on the following conditions:

If *2nd_string* is found in *1st_string* the location where it was found is returned.

If *start* is a value larger than the length of *1st_string*, 0 is returned.

If the *1st_string* is a null string, 0 is returned.

If *2nd_string* cannot be found, 0 is returned. If *2nd_string* is a null string, *start* is returned; if *start* was not specified, 1 is returned.

Note :As strings in **HiSoft BASIC Professional** have no length limits this function returns a long integer.

• Example

```
'This routine parses the command line for a space to
'find the first parameter passed to the program.
I%=INSTR(COMMAND$," ")
IF I%=0 THEN I%=LEN(COMMAND$)
PRINT "The 1st parameter passed is: ";LEFT$(COMMAND$,I%)
```

INT function

• Syntax

INT(*numeric_expression*)

• Effect

This function returns the largest integer less than or equal to the *numeric_expression*.

• Comments

The value returned is the same type as *numeric_expression*. If you need to find the nearest long integer (i.e. greater than 32767) to a numeric expression use the **CLNG** function.

• Example

```
'This example illustrates the differences between the
'functions CINT, FIX and INT.
PRINT CINT(1.5),CINT(-1.5)
PRINT FIX(1.5),FIX(-1.5)
PRINT INT(1.5),INT(-1.5)
```

Result:

```
2             -2
1             -1
1             -2
```

KILL statement

• Syntax

```
KILL file_spec
```

• Effect

All files fitting the description of *file_spec* are deleted from disk.

• Comments

The *file_spec* must conform to the AmigaDOS format. **KILL** will also delete any icon file associated with *file_spec*.

• Example

```
KILL "DF0:TEMP"
```

LBOUND function

• Syntax

LBOUND(*array*[,*dimension*])

• Effect

This function returns the smallest available subscript of the specified array and optionally of a specific dimension.

• Comments

A common use of **LBOUND** is to determine the size of an array, together with **UBOUND**. *array* is the array to be checked, whereas *dimension* is the number of the dimension to be checked.

In fact because **OPTION BASE** is used for all the subscripts of the array they will all have the same lower bound. However it may be possible to explicitly set the lower bound in the future.

• Example

```
DIM Levels(X,Y,Z)
.
.
PRINT LBOUND (Levels,2)'This will return the lowest
                       'bound for the Y dimension
                       'which is 0 unless otherwise
                       'defined with OPTION BASE
```

LCASE$ function

• Syntax

LCASE$(*string_expression*)

• Effect

This function returns a string with every alphabetic character in lower case.

• Comments

Any characters in the range 'A' to 'Z' inclusive are converted to 'a' to 'z' as required. All other characters are left alone.

• Example

```
PRINT "Do you wish to format your hard disk? Y/N";
DO a$=INKEY$
UNTIL a$<>""
IF LCASE(a$)="y" THEN
        do_format "DH0:"
ELSE
        PRINT "Very wise"
END IF
```

LEFT$ function

• Syntax

```
LEFT$(string_expression,n)
```

• Effect

This function returns a string made up of the leftmost *n* characters of *string_expression*.

• Comments

n is a long integer argument as there is no string length limit in **HiSoft BASIC Professional**. If *n* = 0 then a null string is returned. If *n* is greater than the number of characters in *string_expression*, the entire string is returned.

string_expression may be a string variable, a string literal or a string constant.

• Example

```
String$="HiSoft BASIC Professional"
PRINT LEFT$(String$,6)
```

Result:

```
HiSoft
```

LEN function

• Syntax

LEN(*string_expression*)

• Effect

This function returns the number of characters in *string_expression*.

• Comments

string_expression may be a string variable, a string literal or a string constant. This can return numbers greater than 32767 so be careful if assigning to integers - use long integers instead.

• Example

```
Another_String$="Devpac Amiga by HiSoft"
PRINT LEN(Another_String$)
```

Result:

```
22
```

LET statement

• Syntax

`[LET] variable = expression`

• Effect

Assigns *variable* the value of *expression*.

• Comments

LET is entirely optional and unnecessary; the equals sign is sufficient to assign values to variables.

• Example

```
LET profit=income-sales-tax
```

is the same as

```
profit=income-sales-tax
```

LIBRARY statement

• Syntax

```
LIBRARY library_name[,another_library_name]...

LIBRARY CLOSE
```

• Effect

This statement defines which Amiga libraries are to be used by the program and opens that library. **LIBRARY CLOSE** is used to close all open libraries.

• Comments

To open a library you will need a `.bmap` file when *compiling* your program. When your program is running the corresponding .library file is required if the library is not built-in to the ROM.

HiSoft BASIC Professional uses the same format for `.bmap` files as AmigaBASIC. See your AmigaBASIC manual for details. If a drive or directory specifier is used then the compiler will look for the `.bmap` file there; if one is not then the current directory and then the `Libs:` directory will be searched. The .name of the `.bmap` file is produced by removing `.library` from the end of the library name and replacing it with `.bmap`.

Any drive or directory name will not be used when the library is opened by the program when it runs. Disk-loaded libraries are normally stored in the `Libs:` directory.

• Example

```
LIBRARY "diskfont.library"
```

LINE statement

• Syntax

```
LINE [[STEP](x1,y1) ] - [STEP] (x2,y2)[,colour],[b[f]]
```

• Effect

Draws a line, box or filled box in the current window.

• Comments

In the simplest case this statements draws a line between the point (*x1*,*y1*) and (*x1*,*y2*). If the co-ordinates (*x1*,*y1*) are omitted then the line is drawn from the current graphics drawing position.

If the `STEP` keyword is used then the following co-ordinate is treated as relative to the current graphics drawing position rather than relative to the current window.

The *colour* parameter specifies the colour that the line will be drawn in; if it is omitted then the current foreground colour is used (this can be set using the **COLOR** statement).

If the final parameter is `,B` or `,b` then an empty box with the co-ordinates giving the opposite corners of the box. If the final parameter is `,BF` then a filled box is drawn; again with the co-ordinates giving the opposite corners.

• Examples

```
LINE (100,100) TO (150,100)
' the above draws a horizontal line from (100,100) to
(150,100)
LINE (100,100) TO STEP (50,0)
' a slightly easier way to achieve the same thing.
LINE -STEP (20,20),2
'draws a diagonal line from the current position in colour
2


LINE (50,50)- STEP (40,0)
LINE -STEP (0,30)
LINE -STEP(-40,-30)
' draw a triangle with corners (50,50),(90,50),(90,80)
```

```
LINE STEP (0,0) - (60,60),,b
'draws a box with corners of (60,60) and the current
graphics position.

FOR i=1 TO 3
        LINE (i*20,i*20)- STEP (15,15), i,bf
NEXT i
' draws 3 filled boxes in three different colours.
```

LINE INPUT statement

• Syntax

```
LINE INPUT[;] ["prompt";] string_variable
```

• Effect

This statement assigns an entire line of input to *string_variable* while ignoring delimiters (such as commas).

• Comments

The *prompt* is printed before input is awaited. A question mark will not be displayed unless it is part of *prompt*. If **LINE INPUT** is immediately followed by a semi-colon, the CR-LF marking the end of the input line will not be echoed to the screen.

LINE INPUT is more useful if you need to enter strings with commas, quotation marks or spaces in them.

• Example

```
LINE INPUT "Enter command:",Com$
```

LINE INPUT# statement

• Syntax

```
LINE INPUT #channel_number,string_variable
```

• Effect

This statement reads a sequence of characters terminated by a CR from the device or file specified by *channel_number* and assigns it to *string_variable*.

• Comments

This statement will read a sequence up to, and including, a CR, and return. A subsequent **LINE INPUT#** will begin reading the second sequence after the CR.

• Example

```
'This routine reads a file in which 5 records that are
'delimited by CRs are kept.
DIM Record$(5)
OPEN "BASE.LST" FOR INPUT AS #1
I%=0
WHILE NOT EOF(1)
        I%=I%+1
        LINE INPUT #1,Record$(I%)
WEND
CLOSE #1
```

LOC function

• Syntax

LOC(*channel_number*)

• Effect

This function returns the program's current position within an **OPEN**ed file.

• Comments

The value returned is a long integer. When used with random-access files, **LOC** returns the number of the last record read or written to the file. For sequential files opened for **OUTPUT**, **INPUT** or **APPEND** the value returned is the number of bytes written or read divided by 128.

• Example

```
SUB WriteBlock
STATIC remember%
remember%=LOC(2)          'get current position
LSET a$=MKI$(remember%)
PUT #2,1                  'use record 1 to hold last
                          'updated record number
PUT #2,remember           'and write actual record
END SUB
```

LOCAL statement

• Syntax

LOCAL *variable_list*

• Effect

This statement declares variables as local to function definitions and sub-programs and creates a new variable each time the function or sub-program is left and re-entered.

• Comments

This statement can be used only within function definitions and sub-programs.

Variables in function definitions usually are global or **STATIC** in sub-programs. The **LOCAL** statement ensures that a new variable is created every time a recursive sub-program or function is called. See **Chapter 4** for more details.

• Example

```
test=10
PRINT FNeasy,test
DEF FNeasy
LOCAL test              'without this the global
test=5                  'test would be changed here
FNeasy=1/test
END SUB
```

LOCATE statement

• Syntax

```
LOCATE [row][,column [,cursor]]
```

• Effect

This statement puts the cursor on *column* and *row* specified; it can also enable or disable the cursor.

• Comments

If any parameter is omitted, **LOCATE** has no effect on that particular value.

column is the column number of the screen where the cursor is to be placed. Valid numbers are 1-62 in 60 column mode, 1-77 in 80 column mode.

row is the row number on the screen where the cursor is to be placed. Valid numbers are 1-25.

The *cursor* parameter is a **HiSoft** extension which if it is 0, the cursor is disabled; if it is 1 then the cursor is enabled.

• Example

```
LOCATE 1,1,0    'this line moves the cursor to
                'the upper left corner of the
                'screen and disables the cursor.
```

LOF function

• Syntax

LOF(*channel_number*)

• Effect

This function returns the length of the file specified by *channel_number*.

• Comments

The value returned for the open file is a long integer.

• Example

```
'This routine opens a file, checks its length, allocates
'enough memory, and reads it in its entirety.
OPEN "LOTSA.INF" FOR INPUT AS #1
Length&=LOF(1)
FileBuf$=INPUT$(Length&,#1)
CLOSE #1
```

LOG, LOG10, LOG2 functions

• Syntax

```
LOG(numeric_expression)

LOG10(numeric_expression)

LOG2(numeric_expression)
```

• Effect

LOG returns the natural logarithm (base *e*) of *numeric_expression*.

LOG10 returns the logarithm base 10 of *numeric_expression*.

LOG2 returns the logarithm base 2 of *numeric_expression*.

• Comments

numeric_expression must be greater than 0.

e is approximately 2.718281828459046

All logarithmic functions are evaluated with single-precision accuracy by default. If the *numeric_expression* is a double-precision number, **LOG*type*** will calculated in double-precision.

• Example

```
PRINT LOG10(10),LOG10(5)
```

LPOS function

• Syntax

LPOS(*argument*)

• Effect

This function returns the position of the printer head.

• Comments

The actual physical position of the printer head is not necessarily returned, as tabs are not expanded. The value returned is the position within the printer buffer starting at 1.

argument is a dummy parameter.

• Example

```
'This routine continues printing strings until the
'printer head reaches the 50th column
WHILE LPOS(0)<50
        I%=I%+1
        LPRINT String$(I%)
WEND
```

LPRINT, LPRINT USING statements

• Syntax

```
LPRINT [expression_list][{;|,}]

LPRINT USING format_string; expression_list [{;|,}]
```

• Effect

These statements print data through the current printer port.

• Comments

LPRINT and **LPRINT USING** are analogous to **PRINT** and **PRINT USING**, except that the data output is via the current printer device, selected with the **Preferences** tool.

• Example

```
LPRINT CHR$(12);        'This line send a form-feed to
                'the device connected to the
                'printer port.
```

LSET statement

• Syntax

```
LSET string_variable = string_expression
```

• Effect

Left-justify a string variable, normally used for **FIELD**ed variables.

• Comments

LSET left-justifies a string variable by padding with spaces on the right up to its length. This is normally used for **FIELD**ed string variables, but can be used with ordinary string variables for formatting output. **RSET** is similar except that it right-justifies.

• Example

```
FIELD #2,20 AS a$
LSET a$=FNGetData$(10)
PUT #2
```

MENU statement

• Syntax

```
MENU title_id,item_id,state[,name_string]

MENU RESET

MENU {ON|OFF|STOP}
```

• Effect

This statement adds and changes the text and state of menu items and titles. **MENU RESET** removes any custom menu items. The **MENU ON**, **MENU OFF** and **MENU STOP** statements control menu event checking.

• Comments

title_id specifies which menu is to be modified with possible values of 1 to 10. 1 is the leftmost menu.

item_id specifies which item on this menu is to be modified unless its value is 0 when it indicates that the title is to be changed.

state may be one of

0	disable menu or title
1	enable menu or title
2	enable the item and place a tick mark beside it.

If the *name_string$* parameter is present then this sets the name of the menu item or title to be that string. If you intend to use a *state* of 2 then you should ensure that the *name_string$* for this item starts with two spaces.

To find out which menu item has been selected you may either wait for a menu to be pressed using the **MENU** function (see below) or use the **ON MENU...GOSUB** statement.

MENU ON should be used to enable menu event checking. This will cause **ON MENU** statements to be acted on whenever an item is clicked.

After a **MENU OFF** statement is executed menu clicks will be ignored (except in the value returned by the **MENU** function).

MENU STOP causes menu events to be stored until a **MENU ON** statement occurs. The **ON MENU...GOSUB** will then be acted on. This can be useful to suspend menu processing whilst some essential code is executing.

See **ON MENU...GOSUB** for a further example.

• Example

```
MENU 1,0,1,"Project"
MENU 1,1,1,"Load"
MENU 1,2,1,"Save"

MENU 2,0,1,"Search"
MENU 2,1,1, "  Find"

MENU 1,2,0      ' disables the Save item
MENU 2,1,2      ' places a tick mark by the Find item
MENU 2,1,1      ' removes the tick again.
```

MENU function

• Syntax

MENU(*n*)

• Effect

This function returns the title and item of the last menu item selected.

• Comments

MENU(0) will normally return 0 unless a menu item has been selected since it was last called, in which case the *title* number (from 1 to 10) is returned. Subsequent MENU(0) calls will return 0 again until another item is selected.

MENU(1) returns the number (from 1 to 19) of the last menu *item* selected.

• Example

```
MENU 1,0,1,"Project"
MENU 1,1,1,"Load"
MENU 1,2,1,"Save"
MENU 1,3,1,"Quit"

MENU 2,0,1,"Search"
MENU 2,1,1, "  Find"

DO
        DO
                title%=menu(0)
        LOOP WHILE title%=0
        PRINT "Title ";title%;" item:;MENU(1)
LOOP UNTIL title%=1 AND MENU(1)=3      'Quit
```

MID$ function

• Syntax

MID$(*string_expression*,*n*[,*length*])

• Effect

This statement returns *length* characters from *string_expression* starting at the *n*th character.

• Comments

If there are fewer than *length* characters in *string_expression*, or if there are fewer than *length* characters to the right of the *n*th character, all characters following the *n*th character are returned. If there are less than *n* characters in *string_expression* , a null string is returned.

• Example

```
Sentence$="This is HiSoft BASIC Professional"
PRINT MID$(Sentence$,9,6)
```

Result:

```
HiSoft
```

MID$ statement

• Syntax

```
MID$(string_variable,n[,length])=string_expression
```

• Effect

Modifies part of a string variable.

• Comments

The characters of *string_variable* starting at position *n* are modified to be the characters of the *string_expression*. If *length* is specified then only this number of characters are replaced; otherwise the whole of *string_expression* is used.

MID$ cannot be used to change the length a string and as a result is more efficient than an equivalent assignment statement. The *string_variable* may be a simple string variable or an element of a string array. Both *n* and *length* may be integers or long integers.

• Example

```
Sentence$="This is HiSoft BASIC Professional"
MID$(Sentence$,1)=" It was"
PRINT Sentence$
```

Result:

```
 It was HiSoft BASIC Professional
```

MKDIR statement

• Syntax

```
MKDIR pathname
```

• Effect

This statement creates the sub-directory as specified by *pathname*.

• Comments

pathname must conform to the AmigaDOS format conventions.

MKDIR can also be used with relative as well as absolute *pathname*s. For a detailed explanation of this, please refer to **CHDIR**.

• Example

```
MKDIR ":ONE"     'an absolute pathname
CHDIR ":ONE"
MKDIR "TWO"      'this is a relative pathname;
                 'it creates :ONE/TWO
```

MKI$,MKFFP$,MKL$,MKS$,MKD$ functions

• Syntax

MKI$(*integer_expression*)

MKFFP$(*single_precision_expression*)

MKL$(*long_integer_expression*)

MKS$(*single_precision_expression*)

MKD$(*double_precision_expression*)

• Effect

These functions convert numeric data of the *expressions* into strings.

• Comments

These functions take a numeric value and store it in a string variable. There is no conversion into human readable form. These functions are often used before outputting numeric data via **LSET** and **RSET**; both of these are only able to process strings. It is vital to note that these functions are not interchangeable with **STR$**. A numeric value processed by a **MK*type*$** function and then printed will be a binary representation of the number.

Although **HiSoft BASIC Professional** uses a different format to the interpreter for single-precision variables, this is transparent to the **MKS$** and **CVS** functions, though a loss of precision (and perhaps range) may occur during the conversion. To avoid this the **MKFFP$** and **CVFFP** functions may be used, though not on data files which have been written or intend to be read by the interpreter.

• Example

```
OPEN "FLD.DAT" AS #1
FIELD #1, 4 AS Lint$    'defines a field
Longint&=65536
LSET Lint$=MKL$(Longint&)       'puts the string value into
                                'the record
PUT #1,1                        'record is written to file
CLOSE #1
```

MOUSE function

• Syntax

MOUSE(*attribute*)

• Effect

Read the current position of the mouse and the status of the left mouse button.

• Comments

The value returned is an integer, the interpretation of which depends on the *attribute*:

0 Status of left mouse button:

- 0 The left mouse button is not down and has not been since the last MOUSE(0) call.
- 1 The left button is not currently down but there has been a single click since the last call to MOUSE(0).
- 2 The left button is currently down but there has been a double-click since the last call to MOUSE(0).
- -1 The user is dragging with the mouse. Strictly the button is currently down and has been pressed once.
- -2 The user is holding the left button down having double-clicked.

1 Current mouse X position.

2 Current mouse Y position.

3 Mouse X position when the last click occurred, before MOUSE(0) was called.

4 Mouse Y position when the last click occurred, before MOUSE(0) was called.

5 Mouse X position when the button was released, if the button was up when MOUSE(0) was called. If the button was down then MOUSE(5) returns the X position when MOUSE(0) was called.

6 Mouse Y position when the button was released, if the button was up when MOUSE(0) was called. If the button was down then MOUSE(6) returns the Y position when MOUSE(0) was called.

• Example

```
DO                          'plot points with left button
        IF MOUSE(0)<>0 THEN PSET(MOUSE(1),MOUSE(2))
LOOP UNTIL INKEY$=CHR$(27)      'until Esc pressed
```

MOUSE statement

• Syntax

```
MOUSE {ON|OFF|STOP}
```

• Effect

Modifies the mouse event trapping to change the effect of **ON...MOUSE** statements.

• Comments

MOUSE ON should be used to enable mouse event checking. This will cause **ON MOUSE** statements to be acted on whenever the user clicks with the left mouse button.

After a **MOUSE OFF** statement is executed menu clicks will be ignored (except in the value returned by the **MOUSE** function).

MOUSE STOP causes mouse events to be stored until a **MOUSE ON** statement occurs. The **ON MOUSE...GOSUB** will then be acted on. This can be useful to suspend menu processing whilst some essential code is executing.

• Example

```
MOUSE STOP
...
MOUSE ON          act on MOUSE events since MOUSE STOP
```

NAME statement

• Syntax

```
NAME old_filename AS new_filename
```

• Effect

This statement renames the file specified by *old_filename* to *new_filename*. It will also rename any icon files.

• Comments

Both file specifications must conform to the AmigaDOS format. It is possible to move files between directories by renaming them. Both arguments are string expressions.

• Example

```
NAME "AFILE.BAS" AS "ANOTHER.BAS"
```

OBJECT statements and functions

• Syntax

```
OBJECT.AX object_id,value
OBJECT.AY object_id,value
OBJECT.CLIP (x1,y1)-(x2,y2)
OBJECT.CLOSE [object_id[,object_id...]]
OBJECT.HIT object_id,[MeMask][,HitMask]
OBJECT.ON object_id[,object_id...]]
OBJECT.OFF object_id[,object_id...]]
OBJECT.PLANES object_id[,plane_pick][,plane_on_off]
OBJECT.PRIORITY object_id,value
OBJECT.SHAPE object_id1,{string_expression|object_id2}
OBJECT.START object_id[,object_id...]]
OBJECT.STOP object_id[,object_id...]]
OBJECT.VX object_id,value
OBJECT.VY object_id,value
OBJECT.VX(object_id)
OBJECT.VY(object_id)
OBJECT.X object_id,value
OBJECT.Y object_id,value
OBJECT.X(object_id)
OBJECT.Y(object_id)
```

• Effect

These are the AmigaBASIC compatible object (or sprite) statements and functions. These are described in detail in the AmigaBASIC manual.

OCT$ function

• Syntax

OCT$(*numeric_expression*)

• Effect

This function returns a string which is the octal representation of *numeric_expression*.

• Comments

OCT$ returns the representation of the integer part of *numeric_expression*. If the expression is an integer then the resulting string will be from 1 to 6 characters in length, but if it is a long integer then it can be up to 11 characters long.

• Example

```
PRINT OCT$(8)
```

Result:

```
10
```

ON...BREAK statement

• Syntax

```
ON BREAK GOSUB {linenumber|linelabel|0}
```

• Effect

Determines the subroutine that is called when `Ctrl-C` or **A.** are pressed.

• Comments

`ON BREAK GOTO 0` causes **HiSoft BASIC Professional** to process break events itself.

BREAK ON should be used to enable break event processing.

• Example

```
ON BREAK GOSUB BreakHandler
BREAK ON
...
BreakHandler:
        RETURN  ' does nothing and thus ignores]
                ' the break keys.
```

ON...COLLISION statement

• Syntax

```
ON COLLISION GOSUB {linenumber|linelabel|0}
```

• Effect

Determines the subroutine that is called when an object collides with the border or another object.

• Comments

ON COLLISION GOTO 0 causes **HiSoft BASIC Professional** to ignore collision events.

COLLISION ON should be used to enable collision event processing.

• Example

```
ON COLLISION GOSUB CollisionHandler
```

ON...ERROR statement

• Syntax

ON ERROR GOTO {*linenumber*|*linelabel*|*0*}

• Effect

Enable error handling and specify error handling routine.

• Comments

This command allows you to trap run-time errors such as overflow or disk full and pass control to a specific BASIC line if any errors occur. The error handler *must* be at the main level of your program - it cannot be within a sub-program or function.

If a line number of 0 is specified, any error handling is disabled and subsequent errors will abort program execution in the normal way. If **ON ERROR GOTO 0** is specified within an error handler it will cause the original error message to be printed and program execution will be halted.

If you wish program execution to continue after an error you should use the **RESUME** statement.

Use of **ON ERROR** in a program causes a larger program size and slightly slower execution speed due to the saving of extra information while a program runs.

• Example

```
ON ERROR GOTO handler
OPEN "DATA.INF" FOR INPUT AS #2
INPUT LINE #2,a$
CLOSE #2
PRINT "Data is ";a$ : STOP
handler:
IF ERR=53 THEN
        PRINT "Error - File DATA.INF not found" : STOP
ELSE
        ON ERROR GOTO 0 'another error so report
END IF
```

ON...GOSUB statement

• Syntax

```
ON n GOSUB {linenumber|linelabel} [,{linenumber|linelabel…}]…
```

• Effect

This statement calls one of a list of subroutines depending on the value of a parameter.

• Comments

If *n* has a fractional part it will be rounded to an integer.

n determines which *label* is jumped to. If *n* is 1 then the first subroutine is **GOSUB**ed, else if *n* is 2 then the second label is the the subroutine that is executed, and so on for all the given subroutines.

If *n* is less than 1 or greater than the number of subroutines available, program execution continues after the **ON...GOSUB** statement.

• Example

```
ON Number% GOSUB One-routine, Two_routine, Three_routine
'If Number% = 3 then Three-routine will be executed.
```

ON...GOTO statement

• Syntax

```
ON n GOTO {linenumber|label} [,{linenumber|label}]...
```

• Effect

This statement causes program execution to branch to one of a list of program lines depending on the value of a parameter.

• Comments

This statement differs from **ON...GOSUB** in that program execution does not return to the line after this statement.

If *n* has a fractional part then it will be rounded to an integer.

n determines which label is jumped to. If *n* is greater than the number of labels specified, or if *n* is less than 1, program execution continues after the **ON...GOTO** statement.

• Example

```
ON Number% GOTO One_branch, Two_branch, Three_branch
'If Number% = 1 then program execution will continue at
'One_branch.
```

ON...MENU statement

• Syntax

ON MENU GOSUB {*linenumber|linelabel|0*}

• Effect

Determines the subroutine that is called when the user clicks on a menu item.

• Comments

ON MENU GOTO 0 disables the menu event.

MENU ON should be used to enable menu checks. Use the **MENU** function to find which Menu item has been selected.

• Example

```
MENU 1,0,1,"Project"
MENU 1,1,1,"Load"
MENU 1,2,1,"Save"
MENU 1,3,1,"Quit"

MENU 2,0,1,"Search"
MENU 2,1,1, "  Find"

ON MENU GOSUB MenuHandler
MENU ON 'enable MENU events
'The main program would go here
SLEEP   'Wait for an event to happen

MenuHandler:
        title%=menu(0)
        PRINT "Title ";title%;" item:;MENU(1)
        IF title%=1 AND MENU(1)=3 THEN STOP     'Quit
        RETURN  'Quit
```

ON...MOUSE statement

• Syntax

ON MOUSE GOSUB {*linenumber*|*linelabel*|*0*}

• Effect

Determines the subroutine that is called when the user clicks on the left mouse button.

• Comments

ON MOUSE GOTO 0 disables the mouse event.

MOUSE ON should be used to enable mouse checks. The **MOUSE** function should be used determine the mouse position.

• Example

```
ON MOUSE GOSUB MouseHandler
'The main program would go here
DO
        SLEEP   'Wait for events to happen
LOOP
MouseHandler:
        junk%=MOUSE(0)  'get next mouse event
        PSET (MOUSE(1),MOUSE(1)
        RETURN
```

ON...TIMER statement

• Syntax

```
ON TIMER(n) GOSUB {linenumber|linelabel}

ON TIMER GOSUB 0
```

• Effect

Sets up or disables a timer event.

• Comments

n gives the frequency of timer events in seconds. This is expressed as a single precision floating point number and should be between (0 and 86400).

ON TIMER GOTO 0 disables the timer event.

TIMER ON should be used to enable timer event processing.

• Example

```
ON TIMER(5) GOSUB TimerHandler
'The main program would go here
DO
        SLEEP
LOOP
        'Wait for events to happen

TimerHandler:
        PRINT "Timer events occur every 5 seconds"
        RETURN  'Quit
```

OPEN statement

• Syntax

OPEN *file_spec* [FOR *mode*] AS [#]*channel_num* [LEN=*record_size*]

OR

OPEN *mode_string*, [#]*channel_num*, *file_spec* [,*record_size*]

• Effect

This statement prepares a file for reading or writing.

• Comments

file_spec must conform to AmigaDOS specifications.

mode may be one of the following:

APPEND Specifies a sequential file which is to be appended

INPUT Specifies a sequential file which is to be read from

OUTPUT Specifies a sequential file which is to be written to

RANDOM Specifies a random-access file to be read from or written to

mode_string is a string expression of one character which is the first letter of the *mode* of the file (A, I, O or R), in upper- or lower-case.

channel_num may be any integer value from 1 to 255 inclusive.

record_size specifies the length of each record in bytes in a random-access file, or the internal buffer size for other types of file. The default value for *record_size* is 128 bytes.

OPEN associates a file or device with *channel_num*. This number is used in all read or write operations to access the file or device.

The counterpart to **OPEN** is **CLOSE**. It is a good idea to **CLOSE** a file whenever possible and re**OPEN** it later if necessary. This is a safety measure; if the system should go crazy for some reason, any data which is still in a buffer will be lost and the file may very well later contain garbage.

Normally `file_spec` refers to a disk file, but there are certain device names that may also be used. These names are:

`PAR:` parallel printer port
`SER:` RS232 port
`PRT:` current printer port as chosen with Preferences
`LPT1:` Same as `PRT:`

Note that channels opened with these names will only respond to simple input/output, that is **PRINT#, WRITE# INPUT#** and **INPUT$**. Random access operations and other operations (such as **LOF**) ,that do not make sense on non-disk devices, will not work.

• Example

```
OPEN "RECORDS.RAN" FOR RANDOM AS #1 LEN=32
'This line opens the random-access file RECORDS.RAN for
'reading & writing, associates it with #1 and specifies
'the record length to be 32 bytes.
'the next line does the same using the alternate syntax
'except using #2
OPEN "R",#2,"RECORDS.RAN",32
```

OPTION BASE statement

• Syntax

```
OPTION BASE {0|1}
```

• Effect

This statement defines the lowest subscript value of arrays.

• Comments

OPTION BASE sets the subscript value of the first element in an array. For instance, an `OPTION BASE 1` before `DIM Array%(42)` causes 42 elements (1-42) to be allocated, not the default 43 (0-42).

If the Array Checks option is off when a program is compiled, **OPTION BASE** statements are ignored and **OPTION BASE 0** is assumed.

• Example

```
OPTION BASE 0            'this is default
DIM Another_array%(177) '178 elements will be
                         'allocated, 0 to 177
```

PAINT statement

• Syntax

PAINT [STEP] (*x*,*y*) [,*paint_colour*],[,*border_colour*]

• Effect

Fills an enclosed with a given colour.

• Comments

The area that is filled is based on (*x*,*y*) and must be enclosed with pixels of the colour *border_colour*.

The area is filled with the colour *paint_colour* or in the current foreground colour if this is omitted. The foreground colour may be changed using the **COLOR** statement; the hues of the colours can be changed using the **PALETTE** statement.

IF *border_colour* is omitted then the *paint_colour* is used.

PAINT will leave the current graphics position at the point specified.

This statement may only be used with windows of types 16 to 31(i.e. smart or super-bitmap).

• Examples

```
CIRCLE (100,100),50,3,2
PAINT (100,100),3,2

' draws a filled circle in colour 3 with outside in colour
2.

CIRCLE (100,100),50
PAINT STEP (0,0)
' draws the same circle in the current foreground colour.
```

PALETTE statement

• Syntax

PALETTE *colour_number,red,green,blue*

• Effect

This statement allows you to change the physical appearance of colours on the screen.

• Comments

Each colour on the Amiga can have its actual appearance changed on screen by changing the colour palette. The number of possible colours depends on the number of bit planes that the current screen has and can range from 0 up to 31, although by default only colours 0 to 3 are available.

The *red*, *green*, and *blue* parameters give the proportion of the primary colours for the particular *colour_number*. These proportions may vary from 0 to 1. 0 means none of a particular primary colour, 1 means the maximum amount.

Colour_numbers are often known as *color-ids*.

• Examples

```
PALETTE 1,0,0,0          'make colour 1 black
PALETTE 0,1,1,1          'make colour 0 white

PALETTE 2,1,0,0          'make colour 2 pure red
```

PATTERN statement

• Syntax

PATTERN [*line_pattern*][,*area_array*]

• Effect

Sets the line pattern and/or area fill pattern for graphics.

• Comments

line_pattern is an integer expression which is used as a bit mask for drawing lines. Normally it is best to express this in hexadecimal or binary. See the examples below.

area_array is an integer array that is in a similar manner to *line_pattern* when the **AREAFILL** statement is used. Each item in the array is treated as a 16 bit horizontal mask. The entire array is used; the array should be single-dimensional and have a number of elements that is a power of 2 (e.g. 2, 4, 8, 16 etc). Note that no parentheses are required after the array name.

• Example

```
PATTERN &h5555 ' draw finely dotted lines
PATTERN &b0101010101010101      ' the same expressed in
                                ' binary

PATTERN &9999   ' coarser dotted  lines

DIM PAT%(3)
PAT%(0)=&h8888
PAT%(1)=&h4444
PAT%(2)=&h2222
PAT%(3)=&h1111

PALETTE ,PAT%
' a very "thin" grey fill pattern.
```

PCOPY statement

• Syntax

```
PCOPY
```

• Effect

Dump the current screen to a printer.

• Comments

This has a similar effect as running the ScreenPrint program from the Amiga Extras disk. It uses the printer configuration set by the the **Preferences** tool.

• Example

```
IF a$="C" THEN
        PCOPY                dump screen
END IF
```

PEEK, PEEKB, PEEKL, PEEKW functions

• Syntax

```
PEEK(address)

PEEKB(address)

PEEKL(even_address)

PEEKW(even_address)
```

• Effect

These functions return the contents of the memory specified.

• Comments

PEEK and **PEEKB** return one byte, **PEEKW** returns a word (16-bit) and **PEEKL** returns a long (32-bit). When using **PEEKL** or **PEEKW**, the address specified must be even, otherwise a `Fatal error - unexpected exception` error will occur .

These functions are supplied as fast ways of reading user memory. For this reason, no checks are made as to the even-ness of the addresses for **PEEKL** and **PEEKW**;.

PEEK, PEEKB and **PEEKW** return integers; **PEEKL** returns a long integer.

The **PEEK** function is exactly the same as **PEEKB** in **HiSoft BASIC Professional** on the Amiga. For compatibility with the AmigaBASIC interpreter use **PEEK**; for compatibility with **HiSoft BASIC Professional** on the Atari ST use **PEEKB**.

• Example

```
Seven%=7
PRINT PEEKW(VARPTR(Seven%))
```

Result:

```
 7
```

POINT function

• Syntax

POINT(*x_pixel*,*y_pixel*)

• Effect

This function returns the colour of a particular pixel.

• Comments

x_pixel and *y_pixel* specify the coordinates within the current output window of the pixel to be read. The result corresponds to the table shown under the description of **COLOR**.

If the required pixel is not within the window, a result of -1 is returned.

• Example

```
PRINT "B"
FOR x=0 to 8
FOR y=0 to 16
IF POINT(x,y) THEN LINE (10*(x+1),10*(y+2))-(10,10),b
NEXT y,x
```

POKE, POKEB, POKEL, POKEW statements

• Syntax

```
POKE address, byte_value

POKEB address, byte_value

POKEL even_address, long_value

POKEW even_address, word_value
```

• Effect

These statements write data directly into memory.

• Comments

These statements write either a byte, word or long word to *address*.

POKEL or **POKEW** to an odd address will result in a `Fatal Error - address exception` message.

The **POKE** statement is exactly the same as **POKEB** in **HiSoft BASIC Professional** on the Amiga. For compatibility with the AmigaBASIC interpreter use POKE; for compatibility with **HiSoft BASIC Professional** on the Atari ST use **POKEB**.

• Example

```
Ten%=10
POKEW VARPTR(Ten%),7
PRINT Ten%
```

Result:

```
 7
```

POS function

• Syntax

```
POS(x)
```

• Effect

This function returns the column number of the current cursor position.

• Comments

x is a dummy argument and serves no purpose. The value returned is the column (horizontal position) of the cursor, the leftmost position being 1. *x* should be numeric.

• Example

```
x%=POS(0)
PRINT "The cursor was in column";x%
```

PRESET statement

• Syntax

```
PRESET [STEP] (x_pos,y_pos) [,colour]
```

• Effect

This statement resets or sets a pixel to a given colour in the current window.

• Comments

PRESET is similar to **PSET**, except that if the *colour* parameter is omitted the background colour is used. The parameters *x_pos* and *y_pos* specify the pixel set, relative to the top left of the current window. If the optional **STEP** keyword is specified then the co-ordinates are taken relative to the current graphics position.

• Example

```
CONST ypos=70
DEFINT i
FOR i=10 TO 350
        IF i<300 THEN PSET (i,ypos)
        IF i>50 THEN PRESET (i-50,ypos)
NEXT i
```

PRINT statement

• Syntax

```
PRINT [expression_1][{;| |,}expression_2][;]…
```

• Effect

This statement prints the data defined in the list of expressions on the screen.

• Comments

The list of expressions may contain a series of string and/or numeric expressions. These expressions should be separated with semi-colons, commas or spaces. If the list of expressions does not end in a semi-colon or a comma, a CR will be output.

A **PRINT** by itself will cause a CR.

A semi-colon separating expressions will cause them to be printed without a space between them. A space between parameters has the same effect as a semi-colon. We recommend using a semi-colon. A comma will cause the expression after the comma to be printed at the next tab stop.

SPC ,**TAB** and **PTAB** may be used in **PRINT** statements, but not elsewhere.

Numbers are printed with either a leading space (if positive) or minus sign, and always with a trailing space.

A question-mark may be substituted for **PRINT** if absolutely necessary; this is a feature of some older BASIC interpreters. To enhance compatibility with other BASICs, we have included this option as well.

Note that, for compatibility with AmigaBASIC, the default window width is infinite. So don't use

```
FOR I=1 TO 100
PRINT I,
NEXT I
```

Without using **WIDTH** first; otherwise all 100 numbers will appear on the same line.

• Example

```
PRINT "The range is";1;"to";100,"(inclusive)"
```

Result:

```
The range is 1 to 100   (inclusive)
```

PRINT#, PRINT# USING statements

• Syntax

```
PRINT #channel_num, [USING format_string] expression_list [;]
```

• Effect

These statements write formatted data to a file or device.

• Comments

channel_num is the number specified when the file was opened.

format_string is an optional string of formatting characters. For an explanation of these, please refer to the **PRINT USING** entry.

expression_list is the list of string and/or numeric expressions to be written.

PRINT# writes the data exactly the way **PRINT** would to the screen. For this reason, great care should be taken to format the data written so that it is recognisable for a later **INPUT#**.

• Example

```
FOR I%=1 TO 5
PRINT #1,I%;",";
NEXT I%
```

Result: (in the target file)

```
 1 , 2 , 3 , 4 , 5 ,
```

PRINT USING statement

• Syntax

PRINT USING *format_string*; *expression_list* [{,|;}]

• Effect

This statement prints *expression_list* according to the format specified by *format_string*.

• Comments

expression_list contains the expressions (numeric or string) that are to be printed; they must be separated by semi-colons.

format_string is a string of formatting characters which determine the field and format of the expressions to be printed.

There are three format characters available for formatting strings:

!	This specifies that only the first character of the string is to be printed.
\ \	These specify that 2 + *n* characters of the string are to be printed. *n* is the number of spaces between the backslashes. If the string is longer than 2 + *n*, the extra characters are not printed. If 2 + *n* is larger than the number of characters in the string, the string will be printed left-justified with spaces padding the right.
&	This specifies a variable length field; if the ampersand (&) is the *format_string*, the string is printed without modification.

There are several format characters available for formatting numeric output:

#	The hash represents each digit to be printed. These positions are always filled; if a number has less digits than have been specified, the number will be printed right justified with spaces padding the left.
.	A decimal point is printed. If *format_string* specifies that a digit precedes the decimal, a digit will always be printed, even if is zero.
+	This causes the sign of the number (+/-) to be printed at the beginning or end of the string, depending on whether the + is at the beginning or end of *format_string*.
-	When placed at the end of *format_string*, this causes a minus sign to be printed if the number is negative. If the number is positive, this has no effect.

`**` This causes leading spaces of a number to be filled with asterisks. A double asterisk also represents two more digit positions.

`$$` The double dollar-sign causes a dollar-sign to be printed to the immediate left of the number; two more digit positions, one of which is for the $, are specified.

`**$` This results in a combination of `**` and `$$`; leading spaces are asterisks and a $ is printed before the number. Three more digit positions are specified, one of which is taken up by the dollar-sign. If a number is negative, a minus sign immediately precedes the $.

`,` If specified before a decimal point, a comma is printed every three digits to the left of the decimal. If , is specified at the end of *format_string*, it is printed as part of the number. The comma specifies one additional digit position and has no effect when used together with `^^^^`.

`^^^^` The four carets specify exponential format. Space is reserved for `E+xx` to be printed. the decimal point may be specified to be anywhere in the number. The significant digits of the number are left justified. Unless a leading/trailing + or a trailing - are specified, one digit position to the left of the decimal point will be used to print the number's sign. Note that double-precision numbers can have five digit exponents, so five carets should be used.

`_` An underscore before a character in the *format_string* causes the character to be printed as a literal; an underscore will be printed by specifying two consecutive underscores in the *format_string*.

A percent sign is printed, if the number to be printed exceeds the field in the *format_string*. If a rounded number is larger than the field, a percent sign will be printed in front of the number.

• Example

```
PRINT USING "!";"FRED"
PRINT USING "\ \";"JOHN"
PRINT USING "&";"Today is"
PRINT USING "###.#";25.4
PRINT USING "#.#";2.54
PRINT USING "##.##  ";12.34,5.678,1.2
PRINT USING "+#.# ";-1.1,2.2
PRINT USING "#.#- ";-3.3,4.4
PRINT USING "**#.# ";56.78,-91.2,-3
PRINT USING "$$##.#";123.45
PRINT USING "**$##.##";2.54
PRINT USING "####,.##";4567.89
```

```
PRINT USING "+####^^^^";-0.4444
PRINT USING "_£##.##";4.99
PRINT USING "#.#";33.25
PRINT USING ".##";.999
```

Results:

```
F
JOH
Today is
 25.4
2.5
12.34   5.68   1.20
-1.1 +2.2
3.3- 4.4
*56.8 -91.2 *-3.0
$123.5
***$2.54
4,567.89
-4444E-04
£ 4.99
%33.3
.99
```

PSET statement

• Syntax

```
PSET [STEP] (x_pos,y_pos) [,colour]
```

• Effect

This statement plots a pixel of a given colour in the current window.

• Comments

The *colour* parameter specifies the colour of the pixel to be plotted, and if omitted defaults to the current line colour. The parameters *x_pos* and *y_pos* specify the pixel to be plotted, relative to the top left of the current window. If the optional **STEP** keyword is specified then the co-ordinates are taken relative to the current graphics position.

• Example

```
pi=3.1415926
FOR i=0 to 4*pi STEP 0.05
        PSET (i*50,100+100*SIN(i))
NEXT i
```

PTAB function

• Syntax

```
PTAB(n)
```

• Effect

Moves to a given pixel position whilst printing

• Comments

PTAB is similar to **TAB** except that the position is expressed in pixels rather than characters.

If the current print position is already beyond *n*, then the print position will move to the *n*th column on the next line. If *n* is greater than the output width, the print position is moved to *n* MOD pixel_width. If *n* is less than one, the print position will become 1. **PTAB** may only be used in **PRINT** and **LPRINT** statements.

• Example

```
FOR i%=2 TO 200
        PRINT PTAB(i%);i%;
NEXT i%
```

PUT file I/O statement

• Syntax

```
PUT [#]channel_number[,record_number]
```

• Effect

This statement writes a record from the random-access buffer to the designated random-access file.

• Comments

channel_number is the number specified when the file was opened.

If *record_number* is not specified, the record written will be the next record.

• Example

```
LSET a$=FNaddress$(10)
PUT #1,10
LSET a$=FNaddress$(11)
PUT #1                  'will be record 11 by default
```

PUT graphics statement

• Syntax

```
PUT (x,y),array_name[,verb]
```

• Effect

This statement copies (or *blits*) onto the screen a rectangular image saved with **GET**.

• Comments

The top left of the rectangle relative to the screen is defined by *x* and *y*. If the array was not filled by a **GET** then unpredictable and potentially disastrous events may occur.

The optional *verb* parameter allows the mode of the copy to be specified. This can be in the form of various reserved words, or an integer. Allowed verbs are:

PSET	Copies the image directly onto the screen, obliterating any previous contents.
PRESET	Similar to **PSET** but a negative image results.
AND, OR, XOR	Resulting pixels are a result of the given operation performed on source and destination pixels.

The default verb is **XOR**. The verb may also be described in terms of an integer expression, the effect of which is described in the graphics library of the ROM Kernal Manual, corresponding to the *minterm*.

The height and width that are used are stored when the image is saved with **GET**.

• Example

```
DIM scr%(4347-1)        ' -1 as array starts at 0
GET (10,20)-(100,200),scr%
FOR i%=0 to 200 STEP 20
        PUT (i%,i%),scr%,PSET
NEXT i%
```

RANDOMIZE statement

• Syntax

RANDOMIZE [*expression*]

• Effect

This statement provides the random number generator with a new seed.

• Comments

If no *expression* is supplied, **RANDOMIZE** will prompt the user for a seed. *expression* may be any integer expression. If **TIMER** is given as *expression*, the **TIMER** function will provide the seed.

The **RND** function will produce the same sequence of random numbers unless re-seeded.

• Example

```
Seed%=42
RANDOMIZE Seed%
Rand!=RND
```

READ statement

• Syntax

```
READ variable_list
```

• Effect

This statement assigns values from **DATA** statements to variables.

• Comments

READ cannot be used without a **DATA** statement. The variable type in a **READ** statement will be converted to the variable type in the **DATA** statement from which the value is read.

The number of elements in a **DATA** statement should not exceed the number of variables in the **READ** statement. If there are more elements in a **DATA** statement then are **READ**, the extra elements are ignored.

To re-**READ** the elements of **DATA** statements, the **RESTORE** statement is used.

• Example

```
DIM Arr&(5)
FOR I%=1 TO 5
        READ Arr&(I%)
NEXT I%
DATA &H42424242,&HFC0000,1200,-45002
```

REDIM statement

• Syntax

REDIM [APPEND] *array(subscripts)*[,*array(subscripts)*]...

• Effect

This statement changes the size of an array.

• Comments

subscripts may be integer or long integer expressions. **REDIM** can even used to change the number of dimensions in an array. Any data that is in an array when it is **REDIM**med is lost.

REDIM APPEND is used to add space to a *one-dimensional* array without destroying the information contained in the array. If the subscript of a **REDIM APPEND** is smaller than the original array, the array will be truncated; this also happens without loss of data in the remaining elements.

• Example

```
DIM Fred%(256)
FOR I%=1 TO 256
        Fred%(I%)=I%
NEXT I%
REDIM APPEND Fred%(260)
PRINT Fred%(42)
```

Result:

```
 42
```

REM statement

• Syntax

```
REM remarks...
```

• Effect

This statements allows remarks to be added within the source, and also to specify metacommands.

• Comments

Comments can also begin with ' (i.e. apostrophe). See **Chapter 4** for details.

If a dollar sign ($) follows the **REM** it is assumed to be a meta-command (see **Appendix A** for further details), else the rest of the line is ignored.

• Example

```
' $OPTION A+
REM The preceding line forces array checks on
```

REPEAT...END REPEAT statement

• Syntax

```
REPEAT name
.
.
[EXIT name]
.
.
END REPEAT name
```

• Effect

The statements within the **REPEAT** loop are executed until an **EXIT** statement for the loop is executed.

• Comments

The *name* is a valid variable name that is not currently being used in the program. You can re-use the same name for a number of different **REPEAT** loops but do not use the name of a sub-program, variable or function.

This loop is the most general looping construct in **HiSoft BASIC Professional** because each loop is named and you can exit out of an outer loop from within an inner one (see the example below). However when this facility is not required **DO...LOOP**s are normally clearer and require less typing.

Note that **REPEAT** is *not* specified in the **EXIT** statement.

• Example

```
i=0: sum=0
REPEAT one
        i=i+1: j=0
        REPEAT two
                j=j+1
                sum=sum+a(i,j)
                IF sum>10000 THEN
                        PRINT "Total too big": EXIT one
                END IF
                IF j=n THEN EXIT two
        END REPEAT two
        IF i=n THEN EXIT one
END REPEAT one
'This example totals the elements in the two-dimensional
'array a() stopping immediately when the sum becomes
'greater than 10000.
```

RESET statement

• Syntax

```
RESET
```

• Effect

This statement closes all open disk files.

• Comments

All disk write buffers are written to disk and the files are then closed.

• Example

```
IF a$="QUIT" THEN RESET : STOP -1
```

RESTORE statement

• Syntax

```
RESTORE [{line_number|label}]
```

• Effect

This statements enables a **READ** statement to access a **DATA** statement that has been read previously.

• Comments

line_number or *label* specify the **DATA** statement that is to be **READ** again. If no *line_number* or *label* is specified, the next **READ** statement will read the first **DATA** statement to appear in the program.

• Example

```
READ John&
PRINT John&
John&=0
PRINT John&
RESTORE J_data
READ John&
PRINT John&
.
.
J_data:
DATA 3448325377,4020987683
```

Result:

```
 3448325377
 0
 3448325377
```

RESUME statement

• Syntax

RESUME {*line_number*|*line_label*}

• Effect

This statement resumes program execution from within an error handling routine at the specified program line.

• Comments

This command is for use from error handling routines set up with **ON ERROR GOTO** to resume execution of the program at the given position. If the statement is used when not in an error-handler a fatal error will occur.

It is the programmers responsibility that the **RESUME** is to the same program level as when the error occurred - that is if it was in a sub-program, function or subroutine the **RESUME** *must* go there for correct program execution. Failure to adhere to this will result in random forms of program crashes.

• Example

```
        a$="TEST.DAT"
open_retry:
        ON ERROR GOTO no_open
        OPEN a$ FOR INPUT AS #1
        ON ERROR GOTO 0 'so that errors are reported
        read data 1
        ......
no_open:
        PRINT "Cannot find file ";a$
        INPUT "Filename:";a$
        RESUME open_retry
```

RETURN statement

• Syntax

```
RETURN [{line_number|label}]
```

• Effect

This statement returns program execution from a subroutine.

• Comments

Program execution is returned to the line after the **GOSUB** statement that called the subroutine. If *line_number* or *label* are specified, execution continues on the specified line.

• Example

```
GOSUB Routine           'Branch to subroutine
.                       'execution continues here
.                       'after the RETURN
Routine:
.
.
RETURN                  'return from subroutine
```

RIGHT$ function

• Syntax

```
RIGHT$(string_expression,n)
```

• Effect

This function returns a string starting at the *n*th character from the right.

• Comments

If *n* is 0, a null string is returned. If *n* is larger than *string_expression*, *string_expression* is returned entirely.

• Example

```
d$=DATE$
PRINT "Year:";RIGHT$(d$,4)
```

RMDIR statement

• Syntax

RMDIR *pathname*

• Effect

This statement removes an empty subdirectory.

• Comments

The subdirectory which is to be deleted must not contain any files. *pathname* must conform to the AmigaDOS conventions.

• Example

```
RMDIR ":fonts"
```

RND function

• Syntax

```
RND[(n)]
```

• Effect

This function returns a pseudo-random single-precision number between 0 and 1.

• Comments

If *n* = 0, the last number generated is returned. If *n* < 0, *n* is used to re-seed the sequence of numbers. If *n* > 0 or is omitted, the next number in the sequence is generated.

The algorithm used is based on that of B.A. Wichman and I.D. Hill (NPL Report DITC 6/82). See also the **RANDOMIZE** statement.

• Example

```
Number!=RND(-1)          'the number returned will have
                         'used -1 as the seed
```

RSET statement

• Syntax

```
RSET string_variable = string_expression
```

• Effect

This statement moves data into a random-access file buffer. It can also be used to right-justify the value of the string in *string_variable*.

• Comments

RSET is the counterpart of **LSET**. If *string_expression* uses less bytes than *string_variable*, the string is right-justified with spaces used as padding.

• Example

```
z$=SPACE$(80)
RSET z$="On the Right of the Screen"
PRINT z$
```

RUN statement

• Syntax

```
RUN [{line_number|file_spec}]
```

• Effect

This statement restarts the current program or the program specified by *file_spec*.

• Comments

If *line_number* is specified, the program is restarted on the line specified. Alphanumeric labels are not allowed.

file_spec can be any valid string expression, and can be any type of executable program. In some BASICs filenames do not need to be in quotes, but they are needed in **HiSoft BASIC Professional**. This facility should not be used when compiling to memory.

RUN will close all open files, windows and screens before executing.

• Example

```
RUN "GENAM2"            'this will start GenAM2 in the
                        'current directory
```

SADD function

• Syntax

SADD(*string_expression*)

• Effect

This function returns the address of *string_expression*.

• Comments

Be very careful when using this function as strings can move about in memory without prior warning if a garbage collect should occur.

• Example

```
Astr$=" BASIC"
PRINT Astr$
POKEB SADD(Astr$),ASC("H")
PRINT Astr$
```

Result:

```
 BASIC
HBASIC
```

SAY statement

• Syntax

```
SAY string_expression[,mode_array]
```

• Effect

Makes the Amiga "speak" given a string of phonemes.

• Comments

mode_array is optional . This array and suitable phonomes are described in detail in the AmigaBASIC manual, so we shall not describe it further here.

The **TRANSLATE$** function is often used in conjunction with the **SAY** statement to avoid the need to construct phonomes directly.

• Example

```
SAY TRANSLATE$("Hello There");
```

SCREEN statement

• Syntax

SCREEN *screen-id,width,height,planes,mode*[,*type*]

OR

SCREEN CLOSE *screen-id*

• Effect

This statement creates or closes an Amiga screen.

• Comments

screen-id should be between 1 and 4 which specifies the screen. This screen-id can be used in subsequent **WINDOW** statements.

SCREEN CLOSE closes the specified screen and any windows that were still open in that screen.

The *width* and *height* parameters give the screen's size in pixels. *planes* gives the number of bit planes associated with the screen so that 1 gives 2 colours, 2 gives 4 colours,3 gives 8 colours, 4 gives 16 colours, 5 gives 32 colours.

If the *type* parameter is omitted then, the *mode* parameter is one of

1 low resolution, non-interlaced

2 high resolution,non-interlaced

3 low-resolution, interlaced

4 high-resolution, interlaced

If the *type* parameter is specified the mode parameter is ignored and the type parameter gives the ViewMode for the screen. These are somewhat complex and are described in detail in the ROM Kernel Volume 1 manual. For example a low resolution HAM (Hold and Modify) screen can be specified with a type of `&h800`.

Note No checks are made on the parameters to the **SCREEN** statement; this has the advantage that you can use add-on screen hardware but the disadvantage that using the wrong numbers can cause very unpredictable effects to occur.

It is very easy to use large amounts of memory when using this statement.

• Example

```
SCREEN 1,640,200,3,2    ' 8 colour high resolution screen.
```

SCROLL statement

• Syntax

```
SCROLL (x1,y1)-(x2-y2),delta_x,delta_y
```

• Effect

Scrolls a region of the current output window

• Comments

(x1,y1)-(x2-y2) specifies the rectangle to be scrolled.

delta_x and *delta_y* give the number of pixels to scroll right and down respectively. Negative values will scroll left and up respectively.

• Example

```
SCROLL (100,100)-(150,150),8,0

' scrolls a 50 by 50 pixel area right up 8 pixels.
```

SELECT...END SELECT statement

• Syntax

```
SELECT [CASE|ON] variable

        [CASE|=] case_list
             [EXIT SELECT]
.
.
        [CASE|=] case_list
.
.
        [CASE|=] [ELSE|REMAINDER]
.
.
END SELECT
```

• Effect

One series of statements is executed depending on the value of *variable*.

• Comments

variable can be any non-subscripted variable, numeric or string. If you wish to select on a more complicated expression then this should be assigned to a temporary variable first.

The *case_list*s consist of the following items separated by commas:

expression

or

first_expression TO *second_expression*

or

relop expression

If the first form is used then if the *variable* is equal to the *expression* then the following statements are executed until the next CASE or = statement.

If the second form is used, then if the *variable* is greater or equal to the *first_expression* and less than or equal to the *second_expression* then the following statements are executed.

In the third form *relop* may be one of =,<>,>,>=,<=,< or == and the following statements are executed if the *variable* satisfies the condition.

If there is a **CASE ELSE**, **CASE REMAINDER**, **=REMAINDER** or **=ELSE** clause then if none of the **CASE**s match then the following statements will be executed. The **CASE ELSE** clause must be the last before the **END SELECT**.

The different alternatives are available for ease of converting programs written in other BASICs. The most common form is **SELECT CASE...CASE...CASE ELSE... END SELECT**.

• Examples

```
INPUT A,B
SELECT CASE A
CASE 1,<0
        PRINT "negative or 1"
CASE >B
        PRINT "greater than B"
CASE 13 TO 19
        PRINT "in the teens"
CASE ELSE
        PRINT "different"
END SELECT
INPUT a$
SELECT a$
="JOHN","FRED": PRINT "John or Fred"
= <"A",>"Z": PRINT "Names must start with a capital letter"
= "MARY": PRINT "Mary"
= REMAINDER
PRINT "unknown name"
END SELECT
```

SGN function

• Syntax

SGN(*numeric_expression*)

• Effect

This statement returns the sign of *numeric_expression*.

• Comments

If *numeric_expression* < 0, -1 is returned.

If *numeric_expression* = 0, 0 is returned.

If *numeric_expression* > 0, 1 is returned.

numeric_expression may be of any numeric type.

• Example

```
Var_one% = -42
Var_two% = 0
Var_thr% = 42
PRINT SGN(Var_one%)
PRINT SGN(Var_two%)
PRINT SGN(Var_thr%)
```

Result:

```
-1
0
1
```

SHARED statement

• Syntax

SHARED *variable_list*

• Effect

This statement enables a subprogram to access variables from the main program without them having been passed as parameters.

• Comments

The *variable_list* may contain variable names and array names, which should be terminated with ().

• Example

```
MAIN:
Fred%=127
John%=76
CALL Routine(John%,10)
.
.
SUB Routine (A%,B%)
SHARED Fred%
Print Fred%,A%,B%
END SUB
```

Result:

```
 127    76      10
```

SIN function

• Syntax

SIN(*numeric_expression*)

• Effect

This function returns the sine of the *numeric_expression* which must be in radians.

• Comments

This function is calculated with single-precision unless the *numeric_expression* is in double precision, thus forcing a double-precision calculation.

• Example

```
DEF FNcosecant(a)=1/SIN(a)
```

SLEEP statement

• Syntax

```
SLEEP
```

• Effect

Causes the program to wait until an event occurs in a "multi-taking friendly" way.

• Comments

The statement will finish if any event that occurs including key-presses, events that have been disabled or suspended using; e.g. **MENU STOP**. For this reason **SLEEP** is often used in a loop.

SLEEP should be used in preference to tight **GOTO**s or **LOOP**s so that other tasks in the system may run.

• Example

```
DO
        IF MOUSE(0)<>0 THEN EXIT LOOP
        SLEEP
LOOP
```

' loops until the left mouse key is clicked.

SOUND statement

• Syntax

```
SOUND frequency,duration,[,[volume][,voice]]

SOUND WAIT

SOUND RESUME
```

• Effect

Produces a sound from the speaker.

• Comments

SOUND WAIT causes subsequent **SOUND** statements to be queued until a **SOUND RESUME** statement is executed; this enables you to synchronise two or more sounds for different voices.

frequency gives the frequency of the sound in Hertz between 20 and 15000. This is a single precision number.

C	130.81	Middle C	523.25
D	146.83	D	587.23
E	164.81	E	659.26
F	174.61	F	701.00
G	196.00	G	738.99
A	220.00	A	880.00
B	246.94	B	993.00
C	261.63	C	1046.50
D	293.66	D	1174.70
E	329.23	E	1318.50
F	349.23	F	1396.90
G	392.00	G	1568.00
A	440.00	A	1760.00
B	493.88	B	1975.50

duration is a single-precision numeric expression between 0 and 77. The time in seconds is *duration*/18.2.

volume is an optional integer parameter from 0 (lowest volume) to 255 (highest volume). If omitted a volume of 127 is used.

voice is an optional parameter giving which Amiga audio channel is being used. The default is 0. Voices 0 and 3 go to the left channel and 1 and 2 to the right.

- **Example**

```
SOUND 523.25,9.1,255,2  ' loud Middle C for half a second
                        ' on the right channel.
```

SPACE$ function

• Syntax

```
SPACE$(n)
```

• Effect

This function returns a string of spaces (ASCII 32), the length of which is specified by *n*.

• Comments

n must be positive, and within the range of long integers. If merely want to print some spaces it is more efficient to use the **SPC** function.

• Example

```
DIM names$(100)
FOR i=0 TO 100
        name$(i)=SPACE$(30)
NEXT i
```

SPC function

• Syntax

SPC(*n*)

• Effect

This function causes *n* spaces to be skipped in a **PRINT** statement.

• Comments

In contrast to **SPACE$**, **SPC** may only be used with **PRINT** and **LPRINT**. *n* must be positive and within the long integer range. It is not necessary to specify a semi-colon after a **SPC**, the function will not produce any other characters except the *n* spaces.

• Example

```
LOCATE 10,1
INPUT "Name:",a$
LOCATE 10,1
PRINT SPC(5+LEN(a$))
LOCATE 10,1
INPUT "Address:",b$
```

SQR function

• Syntax

SQR(*numeric_expression*)

• Effect

this function returns the square root of the *numeric_expression*.

• Comments

The *numeric_expression* must be greater than or equal to 0. **SQR** is normally calculated using single precision maths; if the *numeric_expression* is double-precision, the value returned will be calculated with double-precision.

• Example

```
PRINT SQR(42)
PRINT SQR(42#)

Result:

6.480741
6.48074069840786
```

STATIC statement

• Syntax

STATIC *variable_list*

• Effect

This statement declares variables as local to function definitions and sub-programs and preserves their values when the function or sub-program is left and re-entered.

• Comments

This statement can be used only within function definitions and sub-programs.

Variables in function definitions usually are global. **STATIC** makes these variables local to the specific function.

• Example

```
test=10
PRINT FNeasy,test
DEF FNeasy
STATIC test              'without this the global
test=5                   'test would be changed here
FNeasy=1/test
END SUB
```

STICK function

• Syntax

STICK(*n*)

• Effect

This function returns the x and y positions of the two joysticks.

• Comments

n can be:

0	This returns the x position of joystick 1.
1	This returns the y position of joystick 1.
2	This returns the x position of joystick 2.
3	This returns the y position of joystick 2.

Note: reading joystick 1 will disable subsequent use of the mouse so you have been warned!

For the x positions the function returns:

-1: Left 0: Centre 1: Right

For the y positions the function returns:

-1: Up 0: Centre 1: Down

• Example

```
PSET (200,100)
DO
        XA% = STICK(2)
        YA% = STICK(3)
        PSET STEP (XA%,-YA%)    'plot relatively
LOOP UNTIL STRIG(3)             'until Fire button pressed
```

STOP statement

• Syntax

```
STOP [process_returncode]
```

• Effect

This statement causes the program to end; all files are closed and control is returned to the operating system.

• Comments

If no parameter is specified then the message `Press any key` will appear at the bottom of the screen and it will pause until a key is pressed. To disable this specify a parameter of -1.

If a parameter other than -1 is specified this is used as a AmigaDOS process return code, useful for programs that are to be run from CLIs, for example. By default the return code is 0 (no error), or the error code+300 if a run-time error occurs.

• Example

```
IF FEXISTS(COMMAND$) THEN
        process COMMAND$
        STOP -1          'will return immediately
ELSE
        PRINT "Cannot find ";COMMAND$
        STOP             'will wait for key
END IF
```

STR$ function

• Syntax

STR$(*numeric_expression*)

• Effect

This function returns the string representation of the value of *numeric_expression*.

• Comments

STR$ is the counterpart to **VAL**.

If *numeric_expression* ≥ 0, the string returned contains a leading space, else it will start with a minus sign. *numeric_expression* may be of any numeric type.

• Example

```
A%=4
B%=6
WRITE STR$(A%*B%)
```

Result:

```
" 24"
```

STRIG function

• Syntax

```
STRIG(n)
```

• Effect

This function returns the status of the specified joystick fire button.

• Comments

n may be:

0 This returns -1 if the button of joystick 1 was pressed since the last **STRIG(0)**; 0 is returned if this is not the case.

1 -1 is returned if the button of joystick 1 is currently pressed, 0 if not.

2 This returns -1 if the button of joystick 2 was pressed since the last **STRIG(2)**; 0 is returned if this is not the case.

3 -1 is returned if the button of joystick 2 is currently pressed, 0 if not.

Note: reading joystick 1 will disable subsequent use of the mouse so you have been warned!

• Example

```
IF STRIG(3) = -1 THEN
        PRINT "Button on joystick 2 is being pressed"
END IF
```

STRING$ function

• Syntax

STRING$(*m*, *n*)

STRING$(*m*, *string_expression*)

• Effect

This function returns a string composed of ASCII *n*, which is of *m* length.

• Comments

If *string_expression* is specified instead of *n*, the first character of *string_expression* is used.

• Example

```
PRINT STRING$(40,127)
```

Result:

ΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔ

SUB...END SUB statement

• Syntax

```
SUB global_name[(parameter_list)][STATIC]
.
.
[EXIT SUB]
.
.
END SUB
```

• Effect

This statement is used to define a sub-program.

• Comments

global_name must be a name which is unique throughout the entire program. It is the name which refers to the sub-program.

parameter_list is the list of parameters that are passed to a sub-program. These may be variable names, array names terminated with `()`, or literal values. If preceded by the keyword **VAL** then the parameter is always passed by value; this makes execution of the sub-program quicker.

STATIC is the optional attribute which signifies that the sub-program is not recursive, i.e. it does not call itself. By default, sub-programs have the capability to be recursive. At the time of writing, the compiler ignores this if present.

EXIT SUB causes program execution to return to the part of the program which called the sub-program. It is used to exit the sub-program before the **END SUB**.

Variables within a sub-program may be declared **LOCAL**, **SHARED** or **STATIC**. Please refer to these statements for detailed explanations and also **Chapter 4** for more details of sub-programs.

User defined functions may not be a part of a sub-program definition.

• Example

```
CALL Fred(1,2)
.
.
SUB Fred(One%,Two%)
STATIC RecurDepth
        INCR RecurDepth
        IF RecurDepth > 4 THEN EXIT SUB
        PRINT One% Two%
        CALL Fred(One%*2,Two%*2)
END SUB

Result:

 1  2
 2  4
 4  8
 8  16
```

SWAP statement

• Syntax

```
SWAP 1st_variable, 2nd_variable
```

• Effect

This statement causes the values of the two variables to be exchanged.

• Comments

The two variables must be of exactly the same type, and can be numeric or string.

• Example

```
A% = 1
B% = 2
SWAP A%,B%
PRINT A%;B%

Result:

 2  1
```

SYSTAB function

• Syntax

```
SYSTAB
```

• Effect

Returns the address of an internal system table.

• Comments

This is included for more advanced programmers who may wish to access variables used by the run-time system. The table should be accessed using suitable flavours of the **PEEK** and **POKE** statements.

The table contents are as follows:

Offset	Size	Description
0	word	Workbench screen pixel width
2	word	Workbench screen pixel height
4	long	pointer to Intuition message port
8	long	pointer to Workbench startup message (0 if CLI)
12	long	pointer to screen 1 (0 if closed)
16	long	pointer to screen 2 (0 if closed)
20	long	pointer to screen 3 (0 if closed)
24	long	pointer to screen 4 (0 if closed)
28	long	font pointer to topaz.80
32	byte	number of planes in Workbench screen
33	byte	icon type, used when creating icons for data files, defaults to 2=Project. Changing to 0 disables icon files and stops file operations (e.g. **KILL**) acting on `.info` files
34	long	pointer to be used when creating icon files, normally 0 (for the default icon) but if changed then it is used as a pointer to a suitable image structure

With the exception of the icon entries, these entries are read-only and should *not* be **POKE**d. Undocumented offsets from **SYSTAB** are not guaranteed to remain the same from one revision of the compiler to another but those described above are guaranteed.

- **Example**

```
w%=PEEKW(SYSTAB): h%=PEEKW(SYSTAB+2)
SCREEN 1,w%,h%,PEEKB(SYSTAB+32),2
IF PEEKL(SYSTAB+8) THEN
        a$="From Workbench"
ELSE
        a$="From CLI"
END IF
WINDOW 1,a$,(20,20)-(300,100),,1
```

SYSTEM statement

• Syntax

```
SYSTEM
```

• Effect

This statement causes program execution to end, closes all files, and returns to the operating system.

• Comments

This statement is equivalent to `STOP -1`.

• Example

```
IF UCASE$(a$)="QUIT" THEN SYSTEM
```

TAB function

• Syntax

TAB(*n*)

• Effect

This function causes the print position to move to the *n*th column.

• Comments

If the current print position is already beyond *n*, then the print position will move to the *n*th column on the next line. If *n* is greater than the output width, the print position is moved to *n* MOD WIDTH. If *n* is less than one, the print position will become 1. **TAB** may only be used in **PRINT** and **LPRINT** statements.

• Example

```
'80 column screen
PRINT TAB(-10) "minus ten"
PRINT TAB(175) "one-hundred-and-seventy-five"
PRINT TAB(10) "ten"
```

Result:

```
minus ten
                one-hundred-and-seventy-five
          ten
```

TAN function

• Syntax

TAN(*numeric_expression*)

• Effect

This function returns the tangent of *numeric_expression* which must be in radians.

• Comments

The tangent is normally calculated with single-precision. If *numeric_expression* is double-precision, the value returned will be calculated with double-precision.

• Example

```
PRINT TAN(0.5)
```

TIME$ function

• Syntax

```
TIME$
```

• Effect

This function returns the current system time.

• Comments

A string of the following format is returned:

hh:*mm*:*ss* *hh* is the hours in 24hr format (00-23)
mm is the minutes (00-59)
ss is the seconds (00-59)

• Example

```
PRINT TIME$
```

Result:

```
20:43:56                        'it's 8:43 pm
```

TIMER statement

• Syntax

```
TIMER {ON|OFF|STOP}
```

• Effect

Modifies the timer event trapping to change the effect of **ON...TIMER** statements.

• Comments

TIMER ON should be used to enable timer checking. This will cause **ON TIMER** statements to be acted on whenever a timer event occurs.

After a **TIMER OFF** statement is executed timer events will be disabled until another **TIMER ON** statement is executed.

TIMER STOP causes timer events to be stored until a **TIMER ON** statement occurs. This can be useful to suspend menu processing whilst some essential code is executing.

• Example

```
ON TIMER(60) GOSUB OnceAMinute
TIMER ON
```

TIMER function

• Syntax

```
TIMER
```

• Effect

This function returns the number of seconds since midnight as a single-precision number.

• Comments

This function can be used to provide a seed for **RANDOMIZE**. It returns using 50ths of a second precision. It can be used for timing programs, for example.

Late night programmers beware, examples like the one below will not work if the time between to calls to this function span midnight!

• Example

```
t!=TIMER
call Main
t!=TIMER-t!
PRINT "It took";t!;"seconds"
```

TRANSLATE$ function

- **Syntax**

TRANSLATE$(*string_expression*)

- **Effect**

converts a string from English into phonemes suitable for the **SAY** statement.

- **Example**

```
SAY TRANSLATE$("Hello John")
```

TRON,TROFF statements

• Syntax

```
TRON

TROFF
```

• Effect

These statements allow tracing of a program by line number as it runs.

• Comments

After a **TRON** statement has been executed all program line numbers are printed on the screen as execution occurs. It can be switched off with **TROFF**.

Line numbers are printed within square brackets. If the line does not have a line number the physical line number from the start of the program is printed preface by a comma.

The use of **TRON** in a program considerably slows down program execution, even after a **TROFF** statement. For this reason all **TRON**s should be removed from a program once it has been debugged for maximum performance.

• Example

```
1000 TRON
1010 PRINT "In problem area"
1020 INPUT a
1030 PRINT "Input";a,"Result";fnTest(a)
1040 TROFF
```

UBOUND function

• Syntax

UBOUND(*array_name*[,*dimension*])

• Effect

This function returns the largest available subscript of the array (and optionally the dimension) specified.

• Comments

dimension is specifies the dimension of the array for which the upper bound is to be returned. If *dimension* is not specified the default is 1.

• Example

```
DIM An_array%(10,20,30)
PRINT UBOUND(An_array%,2)
```

Result:

```
 20
```

UCASE$ function

• Syntax

UCASE$(*string_expression*)

• Effect

This function returns a string with every alphabetic character in upper case.

• Comments

Any characters in the range 'a' to 'z' inclusive are converted to 'A' to 'Z' as required. All other characters are left alone.

• Example

```
' the array must be passed by reference, hence VARPTR
' 1 defines the number of dimensions
DEF FN find(a$,VARPTR array$(1))
LOCAL i,f
f=0
FOR i=LBOUND(array$) TO UBOUND(array$)
IF UCASE$(f$)=array$(i) THEN f=i: EXIT FOR
NEXT i
FNfind=f
END DEF
```

VAL function

• Syntax

VAL(*string_expression*)

• Effect

This function returns the numeric value of *string_expression*.

• Comments

This function parses *string_expression* looking for a sequence of characters that can be interpreted as a number. **VAL** will stop reading the string upon finding the first character that cannot be recognized as a number. Leading blanks, tabs, and CR-LFs are ignored. **VAL** returns a double-precision number but it can, of course be assigned to a variable of less precision.

• Example

```
addr$=" 26 Church Lane"
num%=VAL(addr$)          'will be 26
```

VARPTR function

• Syntax

VARPTR(*variable_name*)

or

VARPTR(*#channel*)

• Effect

This function returns the address in memory of the variable *variable_name*, or of the input/output buffer.

• Comments

The address returned is a long integer. **VARPTR** may be used on any numeric variable type; use **SADD** to find the address of strings. To find the address of an array, specify the first element in the array.

Be careful to use **VARPTR** just before the returned address is to be used. Local numeric variables and all strings and arrays may not remain in the same place for the duration of the program.

VARPTR with a channel number as a parameter returns the address of the input/ output buffer for an already **OPEN**ed channel.

VARPTR will always return an even address.

The keyword **VARPTR** is also used in function definitions to denote variable parameters (as opposed to value parameters).

• Example

```
Number% = 7
Addr& = VARPTR(Number%)
PRINT HEX$(Addr&)
```

Result:

```
6A558
```

VARPTRS function

• Syntax

VARPTRS (*subname*)

• Effect

This function returns the address in memory of the sub-program *subname*.

• Comments

The address returned is a long integer. This can be used in conjunction with **CALLS**. The sub-program should have no parameters.

Note: this is for advanced programmers only.

• Example

```
r%=PEEKW(SYSTAB)
SELECT CASE r%
CASE 1: change&=VARPTRS(change_high)
CASE 2: change&=VARPTRS(change_med)
CASE 4: change&=VARPTRS(change_low)
END SELECT
```

WAVE statement

• Syntax

WAVE *voice*,{*wave_array*|SIN}

• Effect

Controls the waveforms used by the **SOUND** statement.

• Comments

voice specifies the audio channel to use.

If the second parameter is **SIN** then the sine function is used to define the waveform. If an array is used it should have at least 256 elements. Elements within the array should be in the range -128 to 127.

• Example

```
WAVE 1,SIN
WAVE 2,wave_table
```

WHILE...WEND statement

• Syntax

```
WHILE condition
.
.
[statements]
.
.
WEND
```

• Effect

A series of statements are executed in a loop until *condition* becomes false.

• Comments

While *condition* is true, the statements between **WHILE** and **WEND** are executed.

WHILE...WENDs may be nested; a **WEND** will match the last **WHILE**.

Be careful not to begin a **WHILE...WEND** loop from within it (e.g. due to a **GOTO**).

DO...UNTIL and **REPEAT** loops are more general versions of the while loop.

• Example

```
A%=0
WHILE A%<10
        INCR A%
        PRINT A%;
WEND
```

Result:

```
 1  2  3  4  5  6  7  8  9  10
```

WIDTH statement

• Syntax

```
WIDTH [#channel_number,] width [,tab_setting]

WIDTH LPRINT width [,tab_setting]

WIDTH string_expression,width [,tab_setting]
```

• Effect

These statements assign a line width to the specified file, screen or printer.

• Comments

The following parameters are valid for this statement:

width	This sets the width for output to the screen.
#channel_number,width	This sets the width of output to *channel_number* to the specified width. Note that the # is compulsory.
tab_setting	sets the length of a tab; default is 14.
string_expression	Is used to set the width and/or tab setting for a device. i.e. one of SCRN,LPT1,COM1. This will affect only channels that are subsequently opened; it does not affect currently open channels.

Specifying **LPRINT** controls the *width* and *tab_length* of any output to the printer.

If a *width* of 255 is specified then the width is set to 'infinite'; that is BASIC never inserts line-feeds during output.

• Example

```
WIDTH 76                        'sets an 80 column screen
WIDTH 61                        'sets a 60 column screen
WIDTH LPRINT 40,8               'set printer output to 40
                                'columns and tab length of 8
WIDTH "LPT1",40,8               'the same
```

WINDOW statements

• Syntax

```
WINDOW OPEN id[,[title_string][,(x1,y1)-(x2,y2)]
        [,[type][,screen_id]]]]

WINDOW CLOSE id

WINDOW OUTPUT id
```

• Effect

Creates and closes windows; **WINDOW OUTPUT** changes the current window.

• Comments

Most commands require a window *id*, which is a number used to identify each window, ranging from 1 to 8 inclusive. By default when program starts it opens window *id* 1 to be a total of 640 by 200 pixels.

All **PRINT, INPUT** and graphics commands use this window, known as the *current window*, until directed otherwise.

WINDOW CLOSE *id*

This closes the window with the given ***id***.

WINDOW OUTPUT *id*

This makes the given window the current one. All **PRINT, INPUT** and graphics commands will go to it.

WINDOW

This is the most complex form of **WINDOW** statement and creates a window of the given specification All parameters (except *id*) are optional and if all are omitted then the existing window is made the active window.

title_string will be used as the window's title bar; the co-ordinates (*x1,y1*) describe the position of the top left of the window and co-ords (*x2,y2*) describe the position of the bottom right of the *interior* of the window. This rather bizarre way of describing windows is compatible with the interpreter but can be overridden (see below). The default rectangle is either a full window based on a 640x200 Workbench screen or the full screen size. The *type* parameter specifies the gadgets associated with the window and is made up by adding different components together as required:

Value	Meaning
1	Window size can be changed using the mouse and sizing gadget
2	Window can be moved around using the title bar
4	Window can be moved in relation to other windows using the normal window gadgets
8	Window can be closed by clicking on the close gadget (will only be noticed if event checks are on, see **Appendix A**)
16	Contents of window re-appear after the window has been covered by other windows. If used in conjunction with type-1 BASIC reserves enough memory for the window to grow to the full size of the screen
32	*Don't* make the window 'gimmezerozero'. This makes window handling much faster, but has the disadvantage that nothing prevents you drawing all over your window borders.
64	Backdrop window; a window that is always at the back of the screen display, like the disk icon display in the Workbench. Makes most sense when used in conjunction with type-32 and type-128 and in a screen other than the Workbench.
128	Borderless window; no border is drawn around it; best used in conjunction with type-32 and -256.
256	What we consider 'sensible' window co-ordinates; using this type changes the interpretation of (*x2,y2*) such that *x2* is the complete window width and *y2* the height.

Types 32-256 are **HiSoft BASIC Professional** extensions, not available in the interpreter. Some experimentation is worthwhile to get the exact type of window you require. The default value is 31.

screen_id refers to a screen created with the **SCREEN** statement and should be from 1 to 4, or -1 to indicate the Workbench screen (the default).

• Example

```
SCREEN 1,640,200,2,2
WINDOW 1,,,16+32+64+128+256,1
LOCATE 10,50
PRINT "I am not the Workbench!"
WINDOW 2,"Ordinary",(0,20)-(320,100),,1
```

WINDOW Function

• Syntax

```
WINDOW(n)
```

• Effect

This function enables certain information to be found about windows that some applications may find useful. The parameter *n* determines the type of result as follows:

Value	Result
0	The window id of the active window, or 0 if the active window is not one of the program's windows.
1	The window id of the current output window
2	The width of the current output window
3	The height of the current output window
4	The x co-ordinate where the next character will be drawn
5	The y co-ordinate where the next character will be drawn
6	The maximum legal colour for the current output window
7	A long integer pointer to the Intuition window for the current output window. If the window has been closed this returns 0.
8	A pointer to the RastPort for the current output window (or 0 if it has been closed).

• Comments

For further details of window pointers and rastports, please see technical documentation regarding Intuition and the graphics library.

The interpreter returns inconsistent results for values of n from 0 to 6 when the window is closed.

WRITE statement

• Syntax

```
WRITE [expression_list]
```

• Effect

This statement prints the data specified in *expression_list* to the screen.

• Comments

This statement is similar to **PRINT**. If *expression_list* is omitted, a line-feed will be printed to the screen. All expressions must be separated by commas. These commas are printed as well. Semi-colon may be used instead of comma; this also outputs a comma.

Strings printed using **WRITE** are quoted, and numbers printed using **WRITE** do not have leading or trailing spaces, unlike **PRINT**.

A line-feed is always output at the end of a **WRITE** statement.

• Example

```
A% = 24
B$ = "STRING!"
PRINT A%,B$
WRITE A%,B$
```

Result:

```
 24             STRING!
24,"STRING!"
```

WRITE# statement

• Syntax

```
WRITE #channel_number, expression_list
```

• Effect

This statement writes the data specified by *expression_list* to a sequential file.

• Comments

WRITE# differs from **PRINT#** in the same ways as **WRITE** differs from **PRINT**. **WRITE#** has the advantage that data can be read back from the file with the equivalent **INPUT#** statement; this is not normally the case for **PRINT**.

For further details, please consult the **PRINT#** and **WRITE** entries.

• Example

```
OPEN f$ FOR OUTPUT AS #1
FOR i=1 TO 10
        WRITE #1,i,CHR$(i+"a"%)
NEXT i
CLOSE #1
OPEN f$ FOR INPUT AS #1
WHILE NOT EOF(1)
        INPUT #1,a,a$
        PRINT a,a$
WEND
CLOSE #1
```

Appendix A
Compiler Options

Meta-Commands

Meta-commands are special compiler commands that cause things to happen during the compilation - they do not produce instructions directly in the compiled code. Meta-commands are specified by following a **REM** statement (or quote ') with a dollar sign.

REM $EVENT ON/OFF

In order for a compiled program to respond to events, that is the statements **ON MOUSE GOSUB, ON MENU GOSUB, ON COLLISION GOSUB, ON TIMER GOSUB, ON BREAK GOSUB** and the closing of windows with the mouse, event checks must be enabled. This increases program size and execution time, but is the price that must be paid for such powerful statements. It is possible to enable and disable these checks as required during the program for maximum efficiency. For example if a complex calculation was being performed in one section of the program it would make sense to place a `REM $EVENT OFF` at its start and a `REM $EVENT ON` at its end. If in the off state then events are remembered but not acted upon, until checks are turned on again.

REM $INCLUDE *filename*

This command lets you include another BASIC source file within the current compilation, just as if it were there in the source. This can be used for using common modules between different programs, or for breaking up larger programs into modules.

REM $OPTION *option_list*

This command lets you specify the various compiler options. Each option is denoted by a letter then, optionally, a + sign to indicate on, or a - sign to indicate off, or a number in the case of some options. Individual options should be separated by commas. For example the line

```
REM $OPTION o+,a-
```

turns overflow checks on and array checks off. Options contained within programs like this override any chosen at the time of compilation, via the Options dialog box. The available options are discussed below.

Compiler Options

HiSoft BASIC Professional has a large range of options that can be used to control various features in compiled programs. There are two ways these options can be specified: in the program itself, using the `$OPTION` meta-command; and at compile time, by clicking on the various radio buttons in the large dialog box.

Array Checks (A)

With array checks on all array accesses are checked to have the correct number of dimensions, and subscripts are checked to ensure they are within the range specified in the **DIM** statement. In addition any reference to an un-dimensioned array will dimension it. For example the program segment

```
DIM a%(10)
a%(20)=10
```

with checks on will produce an `subscript out of range` error. With array checks off there are no checks at all, even on the existence of arrays. If you use an invalid subscript you are likely to destroy something else in memory, and if you try to use un-dimmed arrays you will probably crash the machine, not even getting a Guru.

There is an appreciable speed improvement (though an increase in program size) with checks off, particularly with one-dimensional integer arrays, but this should only be done when your are confident your program works perfectly. Destroying random areas of memory may not be immediately noticeable and may manifest itself some time later in unexpected ways. If you turn array checks off then **OPTION BASE 0** is forced throughout your program.

Break Checks (B)

This option allows a user to break out of a compiled program at certain times. With the option on `Ctrl-C` or `Amiga-period` will abort a running program. Checks are made during **INPUT** statements, the **INKEY$** function and at the start of each line if event checks are enabled (see above). If you want to allow the program user to break out at other times include a line such as

```
dummy$=INKEY$
```

at suitable places in your program. There is no noticeable speed degradation or program size difference with this option on.

Note that this option will override any **ON BREAK GOSUB** statements.

Error Messages (E)

This option tells the compiler to include within the compiled program a list of textual error messages to be produced after a run-time error occurs. With the option off just a number will be printed in the event of a run-time error, which can be looked up in **Appendix B**. Turning this option off results in a smaller program file if stand-alone mode is used.

Icon File (G)

Normally when a program is compiled to disk an icon is created for it, if one does not already exist. Specifying option `G-` will disable this icon, useful for CLI-type programs. To stop running programs creating icons for their data files you should change the required **SYSTAB** table entry (see **SYSTAB** in **Command Reference**).

Keep Size (K)

This option specifies that a program will initially use a particular amount of memory, the rest left available to the system. If you try to keep more memory than there is available then the error `Insufficient Memory` will appear during program initialisation and execution aborted. The default value is 20k, this area being used for the run-time system globals, global numeric variables and the BASIC heap. The heap itself is used for string variables,I/O buffers and all types of arrays. The minimum value is 10k. For example, to increase the heap size to 40k the line would be

```
REM $OPTION k40
```

Line Numbers (N)

This option adds line number information into your program, which can be very useful while debugging your program. With the option on then after a run-time error the physical line number will be printed and **ERL** will contain the logical line number. With the option on program size is increased by 6 bytes for each non-blank line and there is a resulting degradation in program speed.

Linkable Code (P)

When compiling to disk the compiler normally generates a directly-executable file, but to link with C or assembly-language routines the compiler can generate linkable code. This option is best used in conjunction with the F option, for example

```
REM $OPTION P+,FTEST.OBJ
```

For details of interfacing with other languages, see **Appendix E.**

Output Filename (F)

When compiling to disk the compiler will normally create a file based on the name of the source file, or NONAME if the source has yet to be saved. This option allows an explicit output filename to be specified. This option has no effect if compiling to memory. Unlike other options this must be the last on a line, for example:

```
rem $OPTION O+,FDF1:TEST
```

Overflow Checks (O)

An overflow normally occurs when a numeric value exceeds its specified range. These ranges are shown in the following table:

%	integer	-32768 to 32767
&	long	-2147483648 to 2147483647
!	single	-9.2E18 to +9.2E18
#	double	-1.8D308 to 1.8D308

For example, the statement

```
i%=32000+10000
```

will produce an overflow as 42000 is too large to fit into an integer. With this option on an extra 2 bytes are used for each maths operator and some I/O operations and there is a slight speed degradation. The option should be turned off only after you are sure your program is bug free. Note with checks off the results of calculations that result in overflow are not defined.

Stack Checks (X)

With this option on additional checks are made on the integrity and position of the machine stack. The machine stack is that used for return addresses after **GOSUB**s, function calls and sub-program calls. It is also used for local numeric variables and by the run-time system. With checks on, the stack is checked before a **RETURN** statement is executed to ensure that it is sensible to do so; if not the error `RETURN without GOSUB` will occur. With the option off, no checks are made and a **RETURN** done out of context will cause unpredictable results. If the machine stack should ever run out it will probably cause a Guru, but with checks on the error `Stack overflow` will occur. If you have a heavily recursive program you may need to increase the return stack size, detailed later in the **Changing Stack Size** section.

Stand-Alone Code/Shared Library (L)

This defaults to Shared Library mode (L+) which reduces program size and memory usage. For stand-alone code use the L- option which generates completely stand-alone files, for further details see the **Shared Library vs Stand-Alone** section later.

Symbolic Debug (S)

This option is for those programmers with a knowledge of assembly-language who wish to investigate compiled programs using a low-level debugger, such as **MonAm2** supplied with **Devpac Amiga**. With this option turned on, programs compiled to disk have the run-time system's symbols included within the output file for investigation by debuggers.

Underlines (U)

This is an option to maintain compatibility with some dialects of Microsoft BASIC. With this option on, **HiSoft BASIC Professional** allows underlines in variables and procedure names. With the option off, underlines can be used immediately after a reserved word or identifier, Microsoft style. See **Chapter 5** for more information.

Variable Checks (V)

This option makes the compiler look for undeclared variables, a common source of bugs in BASIC programs. This only works for variables referenced within sub-programs or functions. For example, this program was mis-typed and with variable checks on the programmer will be alerted:

```
SUB initialise(mem%)
SHARED values%()
LOCAL i%

DIM values%(mem%)
FOR i%=1 TO mum%          deliberate mistake
values%(i%)=i%
NEXT i%

END SUB
```

Checks should be normally be on if you are compiling structured programs, but old fashioned BASIC programs should have the option off.

Warnings (W)

If the W+ option is specified then the warning messages normally produced by the compiler will be suppressed.

Window Defeat (Y)

If the Y+ option is specified then the default window will not be produced at the beginning of a program. See the **Defeating Initial Window** section.

Advanced Options

The following options are intended for more advanced programmers to customise a compiled program more exactly to their needs. If you don't understand them then it is unlikely you will need them.

Temporary String Descriptors (T)

In the highly unlikely event of the error `String expression too complex` appearing, the number of temporary string descriptors may be increased from the default value of 15, for examples

```
REM $OPTION T30
```

Maths Stack Size (M)

The maths stack is used for storing temporary numeric values during calculations and function calls. If it should ever run out, the global workspace will be corrupted. The minimum value is 32 bytes and defaults to 256 bytes.

Option Summary

The following tables summarise the compiler options and the letter used to describe them, in the `$OPTION` meta-command.

Once-only Options

Array checks	A
Error messages	E
Icon Files	G
Linkable/executable	P
Output filename	F *filename*
Stack checks	X
Stand-Alone/Library	L
Variable checks	V
Window Defeat	Y

Changeable Options

Break checks	B
Line numbers	N
Overflow	O
Underlines	U
Warnings	W

Once only options have a global effect - that is they are in effect or not for the whole program. *Changeable* options are different because they can be turned on and off through your program, so some areas can have an option on, while others can have it turned off.

Using HiSoft from the CLI

In addition to the integrated, Intuition-based version of the compiler, you can run both the editor and the compiler from the CLI or Shell. This is is intended for users who have committed editor preferences, or who prefer a CLI-type of programming environment.

Using the Editor from the CLI

It is recommended that the editor be renamed to save excessive typing in the CLI, with a line such as

```
rename "HiSoft BASIC Professional" hb
```

It can then be invoked with the command `hb`, optionally followed by the filename to be edited. When invoked from the CLI the editor does not create icon files when saving text files, though the compiler will do so for compiled programs (unless the G option is used). The editor needs to load the compiler during its initialisation, and it searches for the file `HB.Compiler` firstly in the current directory, then in the `C:` directory.

Using the Compiler from the CLI

The compiler may also be renamed, though the editor will not find it if it is,. If using the Shell then an alias would make sense, for example

```
alias hbc hb.compiler
```

The command line takes the form of

```
hb.compiler source_filename [options]
```

The `source_filename` should be the name of the BASIC source file requiring compilation, and `.BAS` is assumed as an extension, but may be set explicitly to be something else. The options should be specified starting with '-' and are the same as those described previously. One important difference between this and the integrated compiler is that all options default to *off* when invoking the compiler directly.

For example, the command

```
hbc DEMO -O+,B+,FTEST
```

will compile the file `DEMO.BAS` with overflow and break checks on and create the output file with the name `TEST`.

Changing Stack Size

By default programs running under both the Workbench and CLI have 4000 bytes of stack but this may on occasions need to be increased. The method depends on the environment under which the program is running:

When run from the Workbench the stack size may changed by selecting the program icon then clicking on `Info` from the `Workbench` menu. The `STACK` entry may then be changed as required.

When run from the CLI the stack size may be changed with the `STACK` statement, before running the program.

If you wish to change the stack size of programs running from the Editor, you will have to change the stack size of the editor itself. This can be done using either of the methods described above. A program may check it's own stack size using `FRE(-2)` at the beginning of the program.

Shared Library vs Stand-Alone

Compiled programs may either use a special library file or include the required parts of the library into the program itself. Which option is required depends on the application itself and the environment under which it is intended to run.

Shared Library (Option L+)

This is the default mode for compiled programs; it means that programs are smaller than true stand-alone programs but they need to access the **HiSoft BASIC Professional** library. This is included on our Disk 1 in the `libs` directory, which you cannot see from the Workbench because it has no icon. It is called `hisoftbasic.library` and is like all other proper Amiga libraries, being re-entrant and can be shared between multiple BASIC programs, including the compiler itself. This option should always be used when compiling to memory from the editor as it saves valuable memory.

Advantages Smaller program files; less memory used when using integrated compiler

Disadvantages Needs additional file; if library is upgraded then a re-compilation may be required

Stand-Alone (Option L-)

This generates true stand-alone programs that do not need the special **HiSoft BASIC Professional** library file described above. The compiler builds in only those parts of the library that are required into the program file itself.

Advantages No additional file needed; independent from library changes

Disadvantages Occupies more disk space; with multiple programs takes more memory space

Note that even stand-alone programs may require some regular Amiga libraries to be present on disk, for example `mathieeedoubbas.library` and `mathieeedoubtrans.library`. To create icon files when using the **OPEN** statement the `icon.library` will also be needed.

Defeating the Initial Window

Normally when a compiled program starts up an initial window is created, which will be the same as that created with the statement

```
WINDOW 1,"Compiled with HiSoft BASIC",(0-0)-(640,200),31+256
```

i.e. a window the exact size of a normal non-PAL Workbench screen with a full complement of gadgets. The default window is not the full size of larger Workbench screens (e.g. PAL) to remain compatible with the interpreter.

Opening your own Initial Window

Specifying the Y compiler option will disable this initial window, which allows a window to exact specification to be opened. If you try to PRINT or INPUT before you have opened your own, either the CLI window will be used (see below) or the error `Bad channel number` will occur, referring to #257. This is because windows are referred to internally with channel numbers starting at #257 for window 1.

Creating CLI-type Programs

Another use for this option is to create programs to run from the CLI. If this option is used in a program run from the CLI then all **PRINT** and **INPUT** statements will take place via the CLI window and any indirection will be respected. Note that the BASIC CLI window driver is (deliberately) very simple; statements like **CLS**, **LOCATE** and all graphics commands will give run-time errors; if you need to position the cursor or clear the screen then suitable ANSI codes should be used. The **INKEY$** function is not recommended due to the way the console driver buffers input a line at a time.

If break checks are on and `Ctrl-C` is pressed during console I/O (or a `BREAK` command has been issued by another CLI) then the program will terminate. If your CLI-type program has an icon file is it possible to set up an automatic window for it when it is run from the Workbench. To do this add a `WINDOW` Tool Type specification, for example

```
WINDOW=CON:0/0/640/200/Automatic Windows!
```

This is how the `HB.Compiler` program gets its CLI-type window.

Appendix B
Error Messages

AmigaDOS Error Numbers

This appendix details the AmigaDOS error numbers and their meanings.

103	insufficient free store	out of memory
104	task table full	limit of 20 CLIs
120	argument line invalid or too long	when using CLI commands
121	file is not an object module	trying to execute non-executable file
122	invalid resident library during load	
202	object in use	such as a file by another program
203	object already exists	
204	directory not found	
205	object not found	most commonly a file
206	invalid window	in name specification
209	packet request type unknown	
210	invalid stream component name	name too long or contains ctrl chars
211	invalid object lock	
212	object not of required type	such as directory name instead of file
213	disk not validated	disk is still being validated, or bad
214	disk write-protected	
215	rename across devices attempted	

216	directory not empty	when trying to delete it
218	device not mounted	after specifying a volume name
219	seek error	
220	comment too big	file comments must be less than 80
221	disk full	
222	file is protected from deletion	
223	file is protected from writing	
224	file is protected from reading	
225	not a DOS disk	
226	no disk in drive	
232	no more entries in directory	

Run-time Errors

A run-time error is an error produced by a running program; they are distinct from errors produced by the compiler while compiling your program which are detailed later in the Appendix. There are two types of run-time error: fatal errors, produced when something goes seriously wrong, and ordinary errors.

As a substantial part of the compiler is written in **HiSoft BASIC Professional** it is possible for the compiler to generate these errors.

Fatal Run-time Errors

These are shown in a similar way to a Guru in a red box at the top of the screen, with these ominous words

```
HiSoft BASIC fatal error: <message> <,#line>
Don't panic - press left mouse button
```

The <#line> part of the message will contain the last physical line number, if the line number option was on. Pressing the left mouse button should terminate the program. These are known as *fatal errors* because **ON ERROR** cannot trap them - the problem is considered too serious to continue running the program. Fatal errors that can occur are as follows:

Cannot create HB.mainport

Very unlikely, this means the program's main message port cannot be allocated, normally because memory is extremely low.

Cannot open mathieeedoubbas.library

This library is required for double-precision maths. Incidentally, the Workbench 1.3 version (supplied on Disk 1) is considerably faster than the older version.found on Workbench 1.2 disks.

Cannot open mathieeedoubtrans.library

This library is required for double-precision transcendental routines, including trig functions, and is not present on ordinary Workbench 1.2 disks because it is a new 1.3 library.

Memory List Corrupt

Either there is a bug in the run-time system, or something has corrupted BASICs memory list. It is likely that the machine will crash shortly after you left click, if not sooner.

No hisoftbasic.library

This occurs if a shared library-type program has been run and cannot find the library file. It must be in the LIBS: directory.

Stack overflow

The machine stack has overflowed; see **Changing Stack Size** in **Appendix A**.

String space corrupt

Either something has corrupted the string space (i.e. the heap) or the garbage collector has failed. If this occurs in a repeatable fashion please contact us.

Unexpected Exception

The operating system has tried to put up a `Software error - task held` System Requestor, but BASIC intercepts this. This can be caused by many things, but most common are address errors, caused by accessing odd-aligned memory (e.g. `PEEKL(45)`) and illegal exceptions, caused by stack overflow or memory corruption. If the line number is not a big enough clue as to the cause then a low-level debugger can be used to catch the exact cause of the exception, if the Symbolic Debug option is used.

Non-fatal Run-time Errors

Non-fatal run-time errors are reported in a System Requestor, with a message, module name, line number and channel number, together with an `Abort program` gadget. Some fields will not appear in messages, depending on the compiler options used.

Thee errors are listed in numeric order for easy reference when messages are not included in the compiled program and for the **ERR** function. Following the numeric list there is an alphabetic list with explanations as to their meaning.

3 RETURN without GOSUB

4 Out of data

5 Illegal function call

6 Overflow

7 Out of memory

9 Subscript out of range

10 Redimensioned array

11 Division by zero

13 Type mismatch

16 String formula too complex

20 RESUME without error

31 Wrong number of subscripts

49 Volume not found

50 FIELD overflow

51 Internal error

52 Bad file number

53 File not found
54 Bad file mode
55 File already open
57 Device I/O error
61 Disk full
62 Input past end
63 Bad record number
64 Bad file name
67 Too many files
68 Device function unavailable
70 Disk write protected
75 Path/file access error
76 Path not found
77 Break pressed

Run-time Errors Alphabetically

Bad file mode

You are trying open a channel with an invalid mode string, or trying an input/output operation on a channel that cannot support it, e.g. **PRINT**ing to a file opened for **INPUT**.

Bad file name

The filename is invalid, either because it is too long or because it contains invalid characters.

Bad file number

Valid channel numbers are from 1 to 255 inclusive. In addition windows use channel numbers 257 upwards.

Bad record number

A record number of 0 is not valid in a **PUT** or **GET** statement.

Break pressed

`Ctrl-C` or Æ-period was pressed This error cannot be trapped by **ON ERROR** but can with **ON BREAK** - if you wish `Ctrl-C` to be ignored then turn the break checks option off.

Device function unavailable

You are trying an operation on a device that cannot support it, such as drawing a circle on a disk file.

Device I/O error

A physical error has occurred during an input/output operation.

Disk full

Disk write protected

Division by zero

FIELD overflow

A **FIELD** statement has attempted to use more space than specified in the record size when **OPEN**ed, or **PRINT#** to a random file has filled the current record.

File already open

Another program has probably opened the file.

File not found

Illegal function call

This error can be caused by a multitude of things and means a parameter was not in the range required by a function. The module name and line number can be used to track down the error.

Input past end

You have tried to read past the end of a file.

Out of data

A **READ** has been attempted when no more **DATA** is available.

Out of memory

Normally this means the heap is full, which is the area of memory used for storing strings and arrays. This can be increased by the use of the **Keep** option described in **Appendix A**. It can also mean that the system has run out of memory while BASIC was trying to allocate, for example, some chip memory to do a graphic operation.

Overflow

A result of a numeric calculation is too large to fit into the required type. This can occur when you don't expect it, for example the line

```
test!=32767+1
```

will produce it, as two integers are added together using integer arithmetic. The correct result can be obtained by forcing single-precision arithmetic:

```
test!=32767!+1
```

Path not found

An invalid path was specified in a DOS command.

Path/file access error

You are not permitted to do that operation with that file, such as trying to **KILL** a read-only file.

Redimensioned array

An existing array must be **ERASE**d before **DIM**med for a second time.

RESUME without error

A **RESUME** command will only work from within an **ON ERROR** handler.

RETURN without GOSUB

A **RETURN** statement was executed but not from a section of the program which had been called with **GOSUB**. This error will only occur with the stack checks option turned on (see **Appendix A**).

String formula too complex

String operations use temporary descriptors for intermediate results, and it is remotely possible to run out and produce this error.

Subscript out of range

An array subscript used is larger than that specified in the dimensioned statement, negative, or 0 if `OPTION BASE 1` is used.

Too many files

A maximum of 255 channels may be opened at once from BASIC.

Type mismatch

A **READ** was attempted into a numeric variable but the data was found to be a string.

Volume not found

Wrong number of subscripts

An array has been referenced with a different number of dimensions to the number last **DIM**med. Unlike other compilers this is a run-time error as **HiSoft BASIC Professional** arrays can be **REDIM**med with different numbers of dimensions.

Compilation Errors

When the compiler detects an error or something that may be an error (a warning) it generates a message.

The message is prefaced by the error number and followed by the line and file in which the error was detected. In the case of warnings the compiler will continue automatically. After an error it will ask you if you wish to continue. If you type `n` or `N` (for no) you will be returned to the editor. If you hit any other key compilation will continue.

When you return to the editor the cursor will be positioned on the line of the first error and the error message will be displayed in the status line; you can use `Alt-J` to move to the next error. If you have a large number of errors the editor may not be able to remember them all. If you insert or delete lines subsequently, `Alt-J`'s will go to the wrong line. However if you have only inserted one or two lines it should be clear which is the offending line. Simply compile the program again if things get confusing.

Normally the error message should be fairly obvious. The following list gives some extra hints for some of the error messages.

Occasionally the compiler will spot errors somewhat later than you might expect. This is usually because the text up to the point it has read is allowed in certain contexts. If you have missed something out at the end of a line, then the error may be detected at the beginning of the next line. Note that, except in the case of missing sub-programs, line numbers or labels, the error in your program can not be *after* the point where the error was detected.

On occasions the compiler will generate more than one error message as a result of a single error in your program; do not be put off by this. If you get confused, just re-compile.

If you have a very badly formed source file, the compiler may slow down considerably. It is probably a good idea to type `n` to the continue prompt.

It is possible that some error messages may have been added to the compiler; only if an error is generated without an error message is this a possible bug in the compiler. If you do get an error number without a message, please tell us; see **Appendix I**

Compiler Error Messages

In the following list many of the messages give an identifier, reserved word or symbol, this is displayed instead of the **%** in the message.

2 ON... should be followed by GOTO/GOSUB

May occur if the expression is misformed.

3 Undefined identifier %

Occurs within sub-programs and functions if the **V** option is on. See **Appendix A.**

4 Unterminated string in data

5 Line number/label missing in GOTO/GOSUB % found instead

This also occurs in **ON GOTO/GOSUB/ERROR** and **RESUME** statements if a line number or label is missing.

6 END DEF assumed

7 END SUB assumed

Error messages 6 and 7 both occur if you attempt to nest sub-program or function definitions.

8 END DEF where END SUB expected

9 END SUB where END DEF expected

Error messages 8 and 9 are both warnings.

10 Letter expected after DEFINT etc

11 Letter expected after - in DEFINT etc

13 Too many ENDs/NEXTs or operand missing

This error is generated in two different contexts, either when you have too many **END**s or **NEXT**s or sometimes when omitting one of the operands to a dyadic operator (e.g. *) in an expression.

14 Numeric expression expected (not string)

Many possible causes.

15) expected instead of %

Many possible causes.

16 Unterminated string

17 Unexpected character in source

This is a warning, the offending character will be ignored.

19 Bad line number (% is not positive)

Or 0 not allowed in this context.

20 Bad line number (% is too large)

21 Bad line number (% has fraction/ exponent)

Or is far too large.

22 Line number % not found

23 Line number % not found

24 Parameter (%) may not be used as FOR loop identifier

Use a local variable instead.

25 Name (not %) expected in parameter list

26 Parameter % appears twice

28) expected at end of parameter list. % found instead

Can also be caused by missing out the comma.

29 Name expected instead of %

Many possible causes.

30 Statement only allowed in sub-program or function

These statements are **LOCAL, SHARED, STATIC, EXIT DEF/SUB.**

32 Label % defined twice

34 Line number/label expected instead of %

Can occur if an expression is badly formed in an **ON...GOTO/GOSUB.**

36 Error during optimisation

37 END FUNCTION expected

39 Internal Error (bad source on pass 2)

Can occasionally be generated by very badly formed source.

40 Expression mismatch

Normally caused by missing operands.

41 END FUNCTION expected before here

42 Expression too complex (too many operators)

More than twenty operators pending.

43 Expression mismatch

44 Expression too complex

More than twenty operands pending.

45 Expression syntax

46 Expression syntax

Normally caused by missing out an entire expression.

47 FUNCTION or SUB expected before here

48 FUNCTION or SUB declared but not present

49 SUB name % used in expression

50 Expression mismatch: % is not a unary operator

51 Expression mismatch: No unary string minus

52 Expression mismatch

53 Illegal type combination

Two strings used with arithmetic operators other than +.

54 Illegal type combination (not string)

A string expression and a numeric expression have been used with an operator.

56 Illegal type combination (not string)

String could not be coerced to numeric type.

57 % is not a variable or the current function

Using other function names is not allowed on the left hand side of assignments.

58 Variable is wrong type

For example, if you pass a double array instead of an integer array to a sub-program.

59 Open bracket expected instead of %

Many possible causes.

60 Comma expected instead of %

Many possible causes. Could be that an expression is mis-typed.

61 Semi-colon expected instead of %

Many possible causes.

62 Extra characters at end of statement % is first

Perhaps you didn't expect the statement to be over so soon. Check your expressions.

63 END DEF assumed at the end of program

64 END SUB assumed at the end of program

65 Code generation failed

This will normally be preceded by one or more code generator errors; for example trying to compile linkable code into memory, or disk full. If you get internal code generator errors please make a note of the line number in the error message. This error will usually be reported at or after the very end of your program.

67 Bad option specified

68 (expected in CALL statement instead of %

When using the **CALL** statement for a sub-program with parameters, you must follow the sub-program name with an open parenthesis.

69 Subroutine % not found

To **CALL** a variable use **CALL LOC**.

72 = expected instead of %

Many possible causes.

78 FOR variables must be simple variables (not % which is an array)

79 FOR variable % can't be a string

80 TO not % expected in FOR statement

Is your initial assignment to the **FOR** variable correct.

83 BASE not % expected after OPTION

84 OPTION BASE must be followed by the number 0 or 1 not %

88 % cannot start a statement

89 Extra ELSE

90 ELSEIF must be followed by THEN not %

91 % not allowed after END

92 Mismatched NEXT should be %

93 END IF expected before here

94 END REPEAT expected before here

95 NEXT expected before here

96 END SELECT expected before here

97 WEND expected before here

98 LOOP/WEND expected before here

Errors 93 to 98 may occur if you have omitted the start of the structured statement in the error line.

99 Name required after SUB or DEF. % found instead

Is it a reserved word? See **Appendix D.**

100 Identifier % redefined

In sub-program or function definition.

102 Name expected after LET (not %)

103 GOTO, THEN or end of line expected after IF expression not %

105 Unexpected END SUB or END DEF

Occurs when not in a function or sub-program definition.

106 CALL missing at start of statement

This is a warning. For more information see **Chapter 5**.

107 Unterminated control constructs

Occurs at the end of sub-programs, user-defined functions and **FOR** loops. Normally means an **END**-something is missing, or a **NEXT**.

108 Function or sub-program redefined in library %

109 Unknown REPEAT loop in EXIT

110 Window statement misformed

111 Semi-colon or comma expected

Many possible causes.

112 REDIM APPEND of arrays of more than one dimension not allowed

113 AS expected instead of %

Occurs in **OPEN**, **NAME** and **FIELD** statements. Perhaps an expression is wrong.

114 CONSTants must be integers not %

115 CONST % can not be assigned to

116 - expected instead of % in graphics GET

118 % is not a label

Your source is probably severely malformed.

119 (expected instead of % in graphics GET or PUT

120 Array expected but ordinary variable % found

121 Argument to VARPTRS must be a subroutine name not %

123 INCLUDE file % not found

125 Value parameter must not be array

126 LOCATE, SOUND or WAVE statement has too many parameters

127 Library must be string literal not %

Have you remembered the quotation marks?

128 Cannot open library

129 Library badly formed

Should never occur.

130 Too many parameters to COLOR statement (max is 3)

131 Sub-program % not found

To **CALL** a variable use **CALL LOC**.

132 GOTO expected after ON ERROR not %

134 SELECT variable must only be a simple variable not %

135 LOOP/WEND expected before here

Appendix C Converting Programs from Other BASICs

Introduction

This Appendix is intended for use by programmers wishing to convert various forms of BASIC into **HiSoft BASIC Professional**. It starts with details about the AmigaBASIC interpreter, then has general notes about conversions followed by sections for particular common BASIC implementations.

AmigaBASIC Compatibility

HiSoft BASIC Professional is designed to compile most existing AmigaBASIC programs unchanged; this section details areas of incompatibilities and possible problems. Note that all references to AmigaBASIC refer to version 1.2, the latest available at the time of writing.

De-tokenising

Before trying to compile any program from the interpreter, the source must be converted to ASCII instead of the special tokenised form that the interpreter uses to store its files. To do this , load the interpreter, then load your program. Select the Output window then type a line such as

```
SAVE "filename.bas",a
```

where `"filename.bas"` should be changed as required. The `,a` at the end is important - it tells the interpreter to save the file as ASCII. We strongly recommend that source files follow the `.bas` naming convention, so you can tell them apart from compiled versions, for example. The file so created can then be used by **HiSoft BASIC Professional** and you can load it back into the interpreter at any time without conversion because **HiSoft BASIC Professional** only uses ASCII files and the interpreter can read them directly.

Unimplemented Features

At the time of writing, the following features of AmigaBASIC are not implemented in **HiSoft BASIC Professional**:

COMMON, RESUME NEXT

In addition, interpreter only statements such as **LOAD**, **SAVE** etc are not implemented as they make no sense in a compiler environment.

Compatibility Issues

We have tried wherever possible to be compatible with the interpreter in both source syntax and run-time emulation.

In terms of source syntax, **HiSoft BASIC Professional** accepts practically all legal AmigaBASIC syntax, except that you may be using a variable which has the same name as one of the additional reserved words, such as **SYSTAB**. When trying to compile existing programs that seemed to run perfectly under the interpreter, the compiler may find errors the interpreter missed, normally in a section of code that seldom gets executed. Note that **HiSoft BASIC Professional** does not allow a variable to be the same name as a program label, but this is unlikely to cause problems.

In terms of run-time emulation, we have tried to emulate the exact actions of the interpreter, where it made sense to do so. We discovered several undocumented features of the interpreter during the development of the compiler (e.g. `FRE(-3)`) and emulated those found. However there are bound to be circumstances that we did not try and if you rely on an undocumented feature it may not work the same under the compiler. There are some features of the run-time system we decided not to emulate, such limiting the maximum window size to non-PAL sized screens and various guru-type bugs.

Note that double-precision variables are stored in a different order in memory between the compiler and the interpreter, but this will only effect programs doing rather nasty things with **VARPTR**. In addition, single-precision numbers are stored in a completely different format (though are still 4 bytes in size) which has less range than that used by the interpreter. When reading and writing random-access files (using **MKD**, **MKS**, **CVD** and **CVS**) we are as compatible as possible with the interpreter though, except for the range limit on single-precision numbers.

Also note that **VARPTR** returns a pointer to a string descriptor, but the string descriptors themselves are very different under the interpreter and the compiler. Programs that use **VARPTR** to directly adjust string descriptors will not work without modification.

Extensions to AmigaBASIC

As well a emulating AmigaBASIC as closely as possible we also extended it in many ways. Amongst the many improvements we have made are:

- Full recursion of sub-programs and functions
- 32-bit hex, octal and binary constants
- Powerful **DO...LOOP** construct
- Much greater control over **WINDOW** types
- **SYSTAB** table for lower-level accessing mode
- **COLOR** statement allows control over drawing mode

Other Conversions

Most BASICs share a certain core of the language, no matter how different they seem to be. **PRINT** always does the same thing, so does **INPUT** and so forth. BASICs can differ by being machine specific, ANSI standard, or conforming to a certain style (MicroSoft's QuickBASIC and Borland's Turbo Basic are notable examples).

Whichever BASIC you are trying to convert a program from, you will need to get the source into ASCII. Some BASICs (e.g. ST BASIC) use ASCII for their programs anyway, but most interpreters use a tokenised form that will come out as gibberish if loaded into the editor. You should save your program from your BASIC interpreter using an ASCII option, if possible.

Some things stand very little chance of being the same from one BASIC to another, in particular internal floating point representations and random-access file formats.

One thing to be aware of particularly is that string length limitations found in other BASICs do not exist in **HiSoft BASIC Professional**. In particular the **LEN** function can return a long integer as a result, not just an integer as in other BASICs.

Old-Style Microsoft BASICs

Microsoft BASIC is the closest thing to a world standard for the BASIC language and is the one we designed **HiSoft BASIC Professional** around.

For this reason most versions of Microsoft BASIC should not present problems converting to **HiSoft BASIC Professional**. Programs written in the old-style BASICs, such as those found in Commodore 64s and under CP/M should require little or no work to convert, as long as machine-specific **PEEK**s and **POKE**s are avoided. You can also save typing by not entering the line numbers that aren't needed.

Most BASIC interpreters from other vendors are at least in part based on the same principles as Microsoft so should also convert reasonably easily.

New-Style Microsoft BASICs

The new style of Microsoft BASIC is defined by QuickBASIC versions 3 and 4 running on IBMs and compatibles. By *new-style* we mean support for structured programming such as sub-programs and parameter passing, **CASE**, **REPEAT**, **DO** etc. The main advantages of **HiSoft BASIC Professional** that you can exploit when converting programs from the IBM world are the large memory and greater graphic support, in particular windows. In addition, recursive programming techniques can be used.

At the present time **HiSoft BASIC Professional** is reasonably compatible with QuickBASIC 3, excluding MS-DOS hardware-specific functions, and has some of the features found in QuickBASIC 4, such as constants and non-FN function names. However **HiSoft BASIC Professional** does not yet support records.

In addition to this high degree of Microsoft compatibility, **HiSoft BASIC Professional** also compiles many of the additional features found in Borland's Turbo Basic compiler for the PC.

Appendix D
Reserved Words

The following is a complete list of the reserved words in **HiSoft BASIC Professional**. These may not be used as sub-program or label names and not normally as variable names. See **Chapter 5** for more details.

ABS	ACCESS	AND	APPEND	AREA
AREAFILL	AS	ASC	ATN	AUTO
BAR	BASE	BEEP	BIN$	BLOAD
BREAK	BSAVE	CALL	CALLS	CASE
CDBL	CHAIN	CHDIR	CHR$	CINT
CIRCLE	CLEAR	CLNG	CLOSE	CLS
COLLISION	COLOR	COMMAND$	COMMON	CONST
COS	CSNG	CSRLIN	CVD	CVFFP
CVI	CVL	CVS	DATA	DATE$
DECLARE	DECR	DEF	DEFDBL	DEFINT
DEFLNG	DEFSNG	DEFSTR	DIM	DO
ELSE	ELSEIF	END	EOF	EQV
ERASE	ERL	ERR	ERROR	EXIT
EXP	FEXISTS	FIELD	FILES	FIX
FOR	FRE	FUNCTION	GET	GOSUB
GOTO	HEX$	IF	INCR	INKEY$
INPUT	INPUT$	INSTR	INT	KEY
KILL	LBOUND	LCASE$	LEFT$	LEN
LET	LIBRARY	LINE	LOC	LOCAL
LOCATE	LOF	LOG	LOG10	LOG2
LOOP	LPOS	LPRINT	LSET	MID$
MKD$	MKDIR	MKFFP$	MKI$	MKL$
MKS$	MOD	MOUSE	NAME	NEXT
NOT	OBJECT.xx	OCT$	OFF	ON
OPEN	OPTION	OR	OUTPUT	PAINT
PALLETTE	PATTERN	PCIRCLE	PCOPY	PEEK
PEEKB	PEEKL	PEEKW	POINT	POKE
POKEB	POKEL	POKEW	POS	PRESET
PRINT	PSET	PUT	RANDOM	RANDOMIZE
READ	REDIM	REM	REMAINDER	REPEAT
RESET	RESTORE	RESUME	RETURN	RIGHT$
RMDIR	RND	RSET	RUN	SADD
SCREEN	SCROLL	SELECT	SGN	SHARED
SIN	SLEEP	SOUND	SPACE$	SPC
SQR	STATIC	STEP	STICK	STOP
STR$	STRIG	STRING$	SUB	SWAP
SYSTAB	SYSTEM	TAB	TAN	THEN
TIME$	TIMER	TO	TRANSLATE$	TROFF
TRON	UBOUND	UCASE$	UNTIL	USING
VAL	VARPTR	VARPTRS	WAVE	WEND
WHILE	WIDTH	WINDOW	WRITE	XOR

Appendix E Assembly Language Details

This Appendix is intended for assembly-language programmers and details the memory maps and register usage of compiled programs together with the creation and use of linkable code. If the previous sentence didn't make any sense then stop reading this Appendix now.

Code Generation

A BASIC source program is converted into true machine-code, there are no P-codes or interpretive run-times. The code produced distinguishes between program area and data area. Program area is what is written to the disk or to memory and is position dependent, relocated either by the AmigaDOS loader in the case of disk files or the code generator itself with programs compiled directly to memory. All program code is ROMmable, executes in user mode and is compatible with 68010, 68020 and 68030 processors.

The figure overleaf shows the overall memory map of a compiled program. Note that the run-time library code will either be in within `hisoftbasic.library` or in a separate code hunk in the program file.

Register Usage

Several registers are committed to special purposes within a compiled program. These are:

A3 - Library Pointer

In order to minimise the space and time taken for run-time library calls, register A3 is dedicated to point to either a run-time jump block (if using the shared library) or to the run-time library code hunk itself, with a $8000 offset to allow a maximum of 64k for the whole library.

Library calls from within the compiled code are of the form

```
JSR       -offset(A3)
```

Library calls from within the library itself do not use A3, they use BSR statements and are resolved by the code generator during compilation in stand-alone mode.

A4 - Local Variable Stack Frame

At the very beginning of functions and sub programs a LINK instruction is done to allocate space on the stack for local variables (and function results) and to establish a register that can be used for accessing parameters. Only space for numeric variables are immediately allocated using LINK; arrays and string descriptors are allocated afterwards using a library call. For example a sub-program which has one local integer variable starts with the instruction

```
LINK      #-2,A4
```

Functions and sub programs finish with a corresponding

```
UNLK      A4
RTS
```

A5 - Data Area Pointer

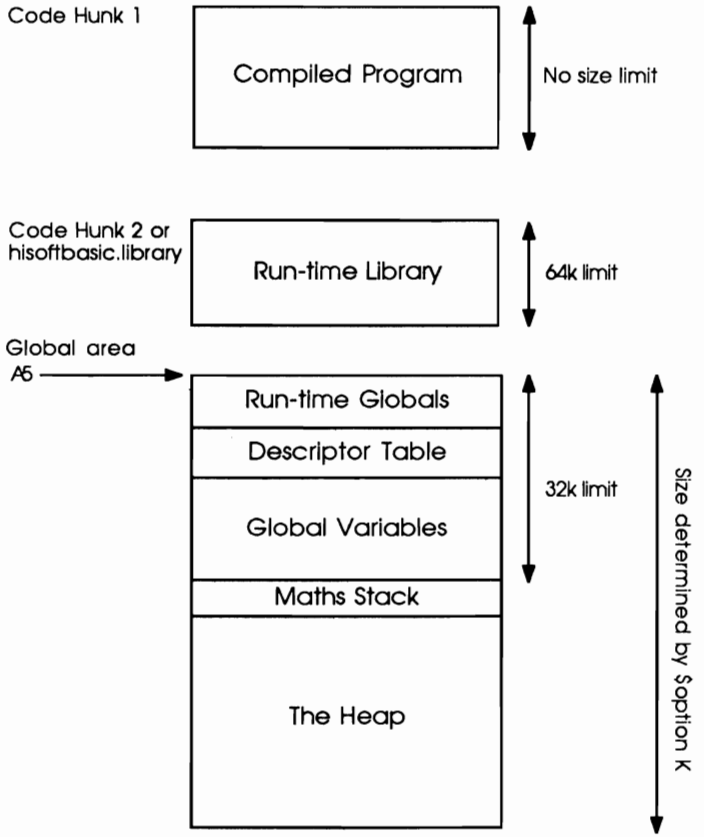
The startup code of a compiled program allocates its global area and sets up A5 to point to its start. At the beginning of the area are the run-time globals, followed by the descriptor table (descriptors are described later). Next is the global variable area, used for storing numeric variables. There is a 32k limit on the total size of these globals, but is would take a massive unstructured BASIC program to require such a number of globals.

A6 - Maths Stack

This is a special stack used for storing intermediate results of numeric calculations.

A7 - Machine Stack

The regular machine stack used for return addresses and local variables (using A4).



Compiled Program Memory Map

Low Level Debugging

It is possible to debug compiled programs at the machine language level using a debugger such as **MonAm2**, supplied with **Devpac Amiga**. If you do this it is strongly recommended that you use the **Symbolic Debug** option when compiling, so that run-time labels will be included in the binary file in the standard AmigaDOS format. In addition this option stops the BASIC run-times from 'stealing' exceptions, so they will be passed onto the debugger.

If the program uses a shared library then the symbols will be of little use to you, though the exception override will be most useful.

Finding Your Way

It can be tricky finding your way around a compiled program and it is recommended that you also specify the **Line Numbers** option when compiling. This option adds instructions of the form

```
MOVE.L    #$AAAABBBB,(A5)
```

to the program. The hex number `AAAA` is the physical line number i.e. the line number displayed by the editor, while the hex number `BBBB` is the last actual line number given in the source. These can be very useful for both finding particular lines and working out where you are in the program.

The first instruction in a compiled program is always a `MOVE.L` instruction setting up A3, followed by a jump around any **DATA** statements. Various registers are then set up before a library call to one of the startup routines. The startup routine itself should *not* be single-stepped, but skipped over. Most `JSR`s can be skipped over, but there is one notable exception to this: `str_constant`. Immediately following it is a string literal and you should not try skipping over the instruction (using `Ctrl-T` in **MonAm**) but single-step the call, then skip over the calls it makes. Alternatively you can set a breakpoint later on in the code after the string in memory.

Note that you may see NOP instructions within the compiled code; don't be alarmed. These are produced by the code generator because on its first pass it leaves room for a JMP absolute instruction when program flow changes, but on pass 2 it notices that the destination is within range for a four byte BRA, so it has to add the NOP (it does not optimise to BRA.S as these take the same time to execute). As the NOPs never get executed there is no speed penalty, but there is a size increase. This is worthwhile; the result is that there are no size limits on compiled programs.

The Heap and Descriptors

Crucial to the operation of **HiSoft BASIC Professional** compiled programs is the area of memory known as the *heap*. BASIC is one of two common languages (the other is LISP) that requires dynamic garbage collection. Owing to the way strings in the language work it is necessary to allocate memory for them as required, then, when it runs out, to re-use all the memory no longer needed. This re-allocation is known as *garbage collection*. Many compilers and even some interpreters do not garbage collect and, as a result, certain operations can cause out-of-memory errors even though there is a lot of unused memory left.

HiSoft BASIC Professional has a very advanced memory management system (which is hidden from the user normally) whereby any memory allocation request can cause a garbage collect to occur in order to satisfy the request. The heap itself is a large block of memory from which allocations for string variables and arrays take their memory.

At the bottom of the heap (low memory) are all the string variables, while at the top are all the arrays. Strings work their way upwards, while arrays grow downwards. Should they ever meet, a garbage collect occurs which deletes all unused strings and moves existing ones around as required. It is important to note that arrays never move in memory as a result of a garbage collect; an array can only move when you REDIM or ERASE any other array.

As items on the heap are liable, without warning, to move about, ordinary pointers are useless.

For this reason strings and arrays are accessed via descriptors, which exist normally in the global area and which themselves contain actual pointers to the data on the heap. As the memory manager knows where all the descriptors are it can update their pointers when a garbage collect occurs.

Incidentally, the garbage collector itself is very fast; for example it can compact around 350k of fragmented heap in under 2 seconds, so there is never any noticeable delay in the running of a program should it have to garbage collect.

Memory Formats

Single-precision Floating Point

These uses the Motorola Fast Floating Point (FFP) format. The format is unusual by most standards, but was designed solely with the 68000 architecture in mind, and is as a result very fast. It is the *not* the same format as used by AmigaBASIC so the **CVS** and **MKS$** functions also do IEEE-FFP conversions.

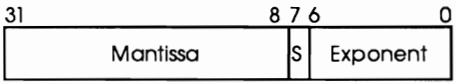


Figure E-2 Single Precision

The sign bit is 0 for positive numbers and 1 for negative numbers. The mantissa has an implied binary point at bit 32 and thus ranges in value from 0.5 to <1.0. The exponent is held in excess -64. The number zero is represented with 32 bits of zero.

Double-precision Floating Point

These use the IEEE format double-precision floats, each occupying 8 bytes, i.e. two longs. Doubles are actually stored in variables in reverse-long order due to the way the `MOVEM` instruction works, which is the opposite way to AmigaBASIC.

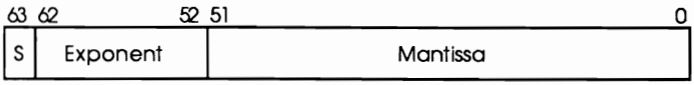


Figure E-3 Double Precision Format

The sign bit is 0 for positive numbers and 1 for negative numbers. The mantissa has an assumed bit of 1; if present it would be at bit 52. The exponent is held in excess-1023. The number zero is represented with 64 bits of zero.

Linkable Code

Although this version of BASIC has many powerful features and extensions, occasionally there is a command or function that is not part of standard BASIC. Additionally, a specific task can be very time critical and need all the speed it can get. For these reasons, **HiSoft BASIC Professional** has the ability to call C or assembly language subroutines.

External Definition

In order to call C or assembly-language the compiler allows linkable code to be generated and this is specified using the P+ option. In addition the program has to declare the names of the routines to call and the number of parameters, using the **DECLARE** statement.

The linkable code generated by the compiler refers to lower-cased version of the names specified (BASIC is case-insensitive) preceded with an underscore character, to be compatible with the popular C compilers for the Amiga. This also helps prevent clashes between C function names and those in the run-time system.

Calling C Functions

C functions are called in a very similar way to the way simple machine-code is called with the **CALL LOC** statement. Any parameters are coerced into long integers and pushed on the machine stack in reverse order (i.e. C order). All registers are destroyable and any return value should be in d0.

Calling Assembly Language Functions

Assembly language functions are called as above, though the assembly-language programmer has to make the function look like a C function. This involves reading parameters from the stack and exporting a lower-case function name starting with an underscore. The code should be assembled to linkable, case dependent code.

Linking

You should end up with a linkable file from the BASIC compiler and a linkable file from the C compiler or assembler. They should be linked together using a linker which supports the standard AmigaDOS format (e.g. ALink or BLink). You may also need to specify C compiler libraries if your C functions have referred to any.

Linked Examples

Here is an artificially simple example of the process, starting with the BASIC program which uses an external function to compare two long integers:

```
REM $OPTION P+,FTEST.OBJ
DECLARE FUNCTION cequal&(p1&,p2&)
PRINT cequal&(1,2),cequal&(10,10)
```

Now the C function:

```
long cequal(a,b)
long a,b
{
return (a==b)
}
```

and the equivalent (Devpac Amiga 2) assembly-language:

```
        opt     l+,c+
        xdef    _cequal
_cequal move.l  4(sp),d0
        cmp.l   8(sp),d0
        seq     d0
        ext.w   d0
        ext.l   d0
        rts
```

Appendix F
Hints and Tips

This chapter shows how you can get the most out of programs written with the **HiSoft BASIC Professional** compiler. It is not necessarily intended only for the advanced programmer.

Using HiSoft BASIC Professional

Making a more efficient program requires knowledge of the features of **HiSoft BASIC Professional**. The following suggestions are intended to give you a firmer understanding of what can be done; we also hope that you will use this information as the basis for a more detailed exploration of **HiSoft BASIC Professional**.

defint a-z

It is a good idea to have this line in your program, it makes the default variable type a short integer. The main benefit from using **int**s is speed. The source becomes more understandable when **&**, **!**, and **#** are used explicitly.

rem $option v+

This forces variable checks on. The primary benefit of this is that you can avoid unnecessary bugs caused by undefined variables in sub-programs and functions.

STATIC variables in SUBs and FNs

STATIC variables retain their values when a **SUB** or **FN** is exited and re-entered. There is a speed benefit in using **STATIC**s as opposed to **LOCAL**s: **STATIC**s are only allocated once, whereas **LOCAL**s must be allocated every time a **SUB** or **FN** is invoked. If your **SUB** or **FN** is recursive then **LOCAL** variables *must* be used.

INCR and DECR

These two statements respectively increment and decrement the value of a variable by one. When used on array elements they are considerably faster than e.g. `Arr(1)=Arr(1)+1`.

==

The double-equals comparison operator has different meanings depending on the type of value compared. When comparing strings, a case-independent comparison is made. It is considerably faster than using **UCASE$** or **LCASE$** and then comparing, or doing something like `IF fred$="A" or fred$="a" THEN...` When comparing numeric values, == is used as 'almost equals'; rounding errors can thus be avoided. There is a performance degradation when using == on numeric values.

! as opposed to

When using floating point maths, it is a good idea to know exactly how much accuracy is actually necessary. Single-precision is much faster than double-precision, due to the degree of accuracy required. If you want floating point, but do not need such a high degree of precision, single-precision is the variable type to use.

VARPTR, SADD and PEEK

When using **SADD**, you must be very careful. The reason for this is quite simple: due to the dynamic heap allocation and the blindingly-fast garbage collection of **HiSoft BASIC Professional**, strings are prone to move around without any prior notice. This is normally completely transparent to the user, but when using these functions it becomes a factor to be reckoned with. If you are going to use them, call the functions just before accessing the variable or array. In addition the address of any array will be invalid if a **REDIM** or **ERASE** statement has been executed since **VARPTR** was used to find the address.

Let us suppose an entire file has been loaded into a single string using **INPUT$**; this is quite possible because strings in **HiSoft BASIC Professional** are only limited by available memory. **SADD** is called to determine where the string is. To go through the file at high speed, all you need is to remember your place in the string and **PEEK** from the location that you need. Do not use **PEEKW** or **PEEKL** because strings can be on odd boundaries and an address exception will occur if you try to read a word or long from an odd address.

Making Your Programs "No-Limits"

If you have been used to programming in more primitive versions of BASIC or in Pascal or C, avoiding arbitrary restrictions on the size and type of data that your program can manipulate can be hard work. For example, having a limit on the length of names that you can type into a business application can sometimes be very annoying. Similarly avoiding limits on the lengths of files and in line lengths can save you a lot of time, if say the file in question has odd end-of-line markers that mean that your program treats the whole file as one line. The following hints should help to avoid this sort of problem:

When reading in or adding to arrays, have code to make the array larger if need be. In general adding a few elements at a time is not a good idea because the program may start spending all its time moving arrays. Normally it is a good idea to start off an array with the size as just larger than a typical requirement, but if an array is only used occasionally then the dimension can start off small. For example, if writing a cross-reference program for **HiSoft BASIC Professional** programs, it would probably a good idea to start by assuming that the number of line numbers is small (say 10) and then if the program turns out to be a horrible old-fashioned program with a line number on every line then the arrays used for holding them can be grown using **REDIM APPEND** at, say, 100 elements at a time.

When using byte or record numbers in files or strings use long integer variables (terminated with **&**). This should remove automatically many 32k or 64k limits on programs that are designed with 16-bit integers in mind.

Appendix G
Bibliography

This bibliography contains our suggestions for further reading on the subject of the Amiga, BASIC, and the operating system. The views expressed are our own and as with all reference books there is no substitute for looking at the books in a good bookshop before making a decision.

BASIC

Amiga BASIC
Published by Commodore-Amiga, Inc.

Yes we know it comes free with your computer but it is a reasonable description of BASIC.

AmigaBASIC Inside and Out
by Data Becker
Published by Abacus

A good book for those who understand the basics (pun intended!) and who wish to learn more about the Amiga-specific features of BASIC. The programming style is rather un-structured though (hardly a SUB to be seen!).

Advanced Amiga BASIC
by Tom R. Halfhill & Charles Brannon
Published by Compute!

To quote the first line, this "is a book for intermediate Amiga BASIC programmers who want to become advanced programmers". It lists many well-written non-trivial BASIC programs as well as shorter examples.

Amiga Technical Manuals

Programmer's Guide to the Amiga
by Robert A. Peck
Published by Sybex

Written with the C or assembly-language programmer in mind it is a much less daunting introduction to the complex operating system than buying all the ROM Kernal Manuals and taking a week off to read them.

ROM Kernal Manuals Volumes 1 and 2
Published by Addison Wesley

These are the official technical manuals, published originally by Commodore-Amiga themselves. They detail the whole operating system (except Hardware and Intuition) but are getting a bit old now. A new edition is expected at the time of writing.

Intuition - The Amiga User Interface
by R.J.Mical and Susan Deyl
Published by Addison Wesley

The definitive guide to Intuition, co-written by the man who designed and implemented it in the first place. The only snag is the current version refers only to the very old version 1.0. Its very readable, containing the memorable programming advice *Dare to be gorgeous and unique! But don't ever be cryptic or otherwise unfathomable. Make it unforgettably great.*

68000

M68000 Programmer's Reference Manual Published by Prentice-Hall

This is the definitive guide to the 68000 instruction set produced by Motorola themselves.

ISBN 0-13-566795-X

68000 Tricks and Traps by Mike Morton BYTE September 1986 issue

By far the best article on 68000 programming we have ever seen. We wish there was a book like this.

Appendix H
Technical Support

So that we can maintain the quality of our technical support service we are detailing how to take best advantage of it. These guidelines will make it easier for us to help you, fix bugs as they get reported and save other users from having the same problem. Technical support is available in four ways:

Phone our technical support hour is normally between 3pm and 4pm, though non-European customers' calls will be accepted at other times.

Post if sending a disk, *please* put your name & address on it.

BIX™ our username is (not surprisingly) *hisoft*. Would UK customers please use CIX or more old fashioned methods; it's cheaper for everyone.

CIX™ our username is (still not surprisingly) *hisoft*.

For bug reports, please always quote the version number of the program (the one displayed in the window title after loading the compiler) and the serial number found on your master disk.

If you think you have found a bug, try and create a small program that reproduces the problem. It is always easier for us to answer your questions if you send us a letter and, if the problem is with a particular source file, enclose a copy on disk (which we will return).

Upgrades

As with all our products, HiSoft BASIC Professional is undergoing continual development and, periodically, new versions become available. We make a small charge for upgrades, though if extensive additional documentation is supplied the charge may be higher. All users who return their registration cards will be notified of major upgrades.

Suggestions

We welcome any comments or suggestions about our programs and, to ensure we remember them, they should be made in writing.

Index

Index

Index

Index

Come and join us at the Roundtable,™ Where the GEnie™ and the Griffin meet!

Does this sound like a fantasy? Well, it may just be a dream come true! When General Electric's high-tech communications network meets MICHTRON's programmers and support crew, ST users around the country will hear more, know more, and save more.

We know that our low prices and superior quality wouldn't mean as much to you without the proper support and service to back them up.

So we are now available on GEnie, the General Electric Network for Information Exchange. GEnie is a computer communications system which lets you use your personal computer, modem, and communication software to gain access to the latest news, product information, electronic mail, games, and MICHTRON's *own* Roundtable (See the special MICRODEAL Section for game information)!!

The Roundtable Special Interest Groups (SIG) gives you a means of conveniently obtaining news about our current products, new releases, and future plans. Messages directly from the authors give you valuable technical support of our products, and the chance to ask questions (usually answered within a single business day).

GEnie differs from other computer communication networks in its incredibly low fees. With GEnie, you don't pay any hidden charges or minimum fees. You pay only for the time you're actually on-line with the MICHTRON product support Roundtable, and the low first-time registration fee.

For more information on GEnie, follow this simple procedure for a free trial run. Then if you like, have ready your VISA, Mastercard or checking account number and you can set up your personal account immediately -- right on-line!

1. Set your modem for half duplex (local echo)--300 or 1200 baud.

2. Dial **1-800-638-8369**. When connected, type **HHH** and press **Return**.

3. At the **U#=** prompt, type **XJM11957,GENIE** and press **Return**.

And don't forget, MICHTRON's Bulletin Board System, The Griffin BBS, is still going strong (the griffin is the half-lion/half-eagle creature on our logo). Our system is located at MICHTRON headquarters in Pontiac, Michigan. For a trial run, call (313) 332-5452.

GEnie and Roundtable are Trademarks of General Electric Information Services.

MichTron®
576 S. Telegraph, Pontiac, MI 48053
Orders and Information (313) 334-5700