# The QUICK Programming Language
USER MANUAL
English Language Version

English Language distribution: Dean Garraghty

# 1. Preface

Welcome to the world of QUICK.

QUICK is an new programming language for your ATARI 8bit Computer. QUICK was written in Germany by Harald Schönfeld and Andreas Binner from 1989 to 1990. The QUICK programming system is published by PPP Germany.

QUICK is almost as fast and powerful as assembler but also as easy to program as BASIC!

This manual will present all features of QUICK. We'll also give you some extra hints, so you can get the most out of QUICK. But, this manual is not intended to explain how to program an 8bit ATARI in general.

## 1.1. Preface to Version 2.x

In 1992 QUICK was revised by Harald Schönfeld once again. Some bugs were removed, the libraries were improved and some examples were included in the manual. Version 2.2 is the latest version. Version 2.1 is also supplied to allow compilation of V1.6 source code.

## 1.2. Preface to the International Version

In Germany QUICK is a big success. Almost everyone doing real programming on the Classic ATARIs knows QUICK and a lot of professional programs are already written in QUICK.

As QUICK is very fast and easy to use, it is perfect for using all hardware features of the 8bit ATARIs. As QUICK is completely software based it has some advantages in comparison with other compiled languages.

Slowly but surely information about QUICK spread out from Germany to “the rest of the world”, and we had more and more questions concerning QUICK from foreign countries.

Finally we translated the manual with the help of Dean Garraghty, and we found distributors for the UK and USA. So, a newly revised version of QUICK is now available.

Have fun!

The authors:

Harald Schönfeld and Andreas Binner

## 2. Introduction

Everyone doing a lot of programming on the 8bit ATARIs will soon find out that it is almost impossible to write efficient programs in ATARI BASIC. It is often too slow and it is impossible to use the unique hardware features of the 8bit machines. To fully use the computer it was necessary to write programs in assembler - but that is much more complicated and takes a lot of time. Already easy things like PRINT, SOUND or BLOAD mean a lot of work and still more debugging. So most programs stay unwritten...

So we decided to write a language that combines the advantages of both BASIC and Assembler: Fast but easy, comfortable and powerful, allowing the use of all hardware features.

So we built in some absolutely unique routines in QUICK, like moving hi-res graphic blocks (BitBlit), playing digitized sounds, easy player movement, automatic DLI and VBI routines, automatic mouse control (ATARI ST-compatible mice) and many more.

It is also possible to do structured programming by using IF-ELSE-ENDIF or REPEAT-UNTIL statements. There are subroutines and it is possible to give parameters to these subroutines. It is possible to define global and local variables of 3 different types: BYTE, WORRD and ARRAY. These ARRAYs can also be used as STRING variables.

As you can see, we did not implement floating point variables (which is one of the reasons for the slow speed of BASIC). But what can you do if you absolutely need such floating point variables? No problem! QUICK is easy to enlarge. Just include the right library!

And there is a final question: Why is QUICK so fast?

QUICK is a compiler based language. That means you first write an ASCII source code which is then compiled by the QUICK compiler. During compilation, libraries can be included, and after adding the runtime file, the compiler produces a stand alone machine code program. It is very important that the compiler is able to translate the QUICK source code in an efficient way into machine code.

## 3. The QUICK Programming System

As you can see, it is always necessary to switch between two different programs: The editor and compiler. Both programs are loaded once, so it is not necessary to load them all the time. This shortens the turn-around-time.

So we need another program, the shell. The shell loads the editor and compiler after booting. It is also a simple user interface which creates the connection between the editor, compiler, and DOS.

The expandability is achieved by libraries. These are source code files containing different subroutines. During compilation, their routines can be added to the program. Four libraries are included on the system disk.

The graphics library provides routines for drawing circles, rectangles and for filling objects. The mathematical and numerical libraries include routines for using the OS floating point variables. The string library provides procedures for copying strings, searching strings, and so on.

Finally the QUICK programming system looks like this:

- Editor for writing the source code
- Compiler for compiling the source code into machine code
- Shell for connecting the editor and compiler
- Libraries for new features

And now - here is the first QUICK source code:

```
MAIN
    PRINT("Hello world!")
ENDMAIN
```

You see, it looks a little like C, Pascal, and BASIC.

Now it's time to see how fast QUICK really is. We wrote a short benchmark program called the “Sieve of Erathostenes”. This is a program to find out the first 1889 primes.

In BASIC you need 363 seconds for this task. In Turbo BASIC you still need 115 seconds, Kyan Pascal only needs 10 seconds, QUICK 5 seconds and ACTION! only 3 seconds. You see, QUICK is faster than most languages for your Classic ATARI and it is the fastest completely software-based programming language of all.

## 4. Installation

QUICK runs on all ATARI 600XL/800XL/800XE/65XE/130XE computers. It does not run on ATARI 1200XL computers with the 1200 OS! QUICK will work best using ATARI DOS 2.5.

Before doing anything with QUICK, you should make a back-up of the system disk. We would like to remind you that you may only copy the QUICK system disk for your own personal use!

On the system disk you will find, among other things, the following files:
- COM compiler
- EXT another part of the compiler
- EDI source code editor
- AUTORUN.SYS shell
- RUNTIME.OBJ runtime file

Also on the disk are ATARI DOS 2.5, demo programs and 4 libraries.

## 4.1. Starting QUICK

Insert the disk in your disk drive and power on the computer while holding down the OPTION key.

After loading DOS 2.5, a ramdisk will be installed on computers with more than 64K of RAM. Then the shell is loaded directly behind DOS, and immediately loads the compiler and editor.

During the development of your programs, you will never need to re-load the editor or the compiler, so you could remove the system disk now. But you will need the library files (*.lib) and the RUNTIME.OBJ file, so these files should be on the disk in drive one.

If you have a ramdisk, the compiler will copy the RUNTIME.OBJ file into the ramdisk during the first compilation, so you don't need this file any more for the following compilations.

If you specify some library names in the source code you don't have to include the *.lib files from drive one. You can store them on any other drive (e.g. the ramdisk D8:), too.

If you work with more than one drive, it will be best to put the system disk in drive one, and put your working disk with your source codes in the other drive.

## 4.2. How to use the Shell

Just press E to start the editor, C to start the compilation, or D to return to the DOS-Shell. Pressing CONTROL-Q will start a cold boot. Pressing RESET will take you back were you came from (hopefully...).

The shell will swap the editor and the compiler in the RAM of your computer. The program that is needed is brought to the free RAM, the other one is stored in RAM at the location of the OS-ROM.

If you still have the system disk in drive one, you can just type C now and you'll enter the compiler. Type D1:QIKDEMO.QIK and a demo program is compiled. After everything is done you can start the program with R. If you want to stop the program, press RESET and you will return to the shell.

You have seen that it takes some time to compile the program. If you use a ramdisk, the compilation will be much, much faster.

The first step in writing your own QUICK programs is to write a source code using the editor.

# 5. How to use the Editor

QUICK's editor is very unlike the BASIC editor as it is not line number oriented. The editor is line oriented. This means you can write up to 38 characters per line. At the end of the line, the cursor stops. Pressing RETURN inserts a blank line beneath the cursor. The editor alway works in insert mode, so characters are inserted in the line, while the characters at the right of the cursor are shifted one column to the right.

At the beginning the editor includes a header, showing the length of the text, the free space and name of the text (as soon as you have saved it once).

On the bottom line you will see “Edit” if you are in the normal edit mode.

## 5.1 Edit Mode

Now you can write your text. You may use capitals or lower case characters, but no inverse characters. Normally all QUICK statements and variables should be written in capitals.

Whenever the editor doesn't react to your typing, you have probably hit the INVERSE or the CONTROL-CAPS key. Just select the key again to clear.

Pressing RETURN will produce an EOL, and the cursor jumps to the beginning of the next line. Pressing RETURN at the end of the line inserts a blank line beneath the cursor. Pressing RETURN in the middle of a line will put the characters at the right of the cursor on to the next line. You can't press RETURN in the 38th column.

Using CONTROL and the cursor keys, you can move the cursor in the text. The start and end position of the text are marked with arrows. You cannot move the cursor over these marks. You also can't move the cursor to the right of the EOL in any line.

BACK SPACE moves the cursor one column left, deletes the character under the cursor and moves the rest of the line to the left. EOL cannot be deleted this way.

CONTROL DELETE deletes the character to the right of the cursor and moves the rest of the line one column to the left. Again, the EOL cannot be deleted.

TAB inserts 2 spaces. You should structure your program as suggested in the demo source codes by using TAB.

ESC switches between text and graphics mode. Press ESC once to insert graphical characters into the text. Press ESC again to return to the normal mode.

During special functions you can use ESC to cancel the function (e.g. exit the save while you type in the filename).

### 5.1.1 The Control Sequences

Pressing CONTROL and one of the following keys will start a special function.

**X Flush -** Deletes the line at the position of the cursor.

**V Unite -** Unites the line at the cursor and the following line. This is only possible if both lines together are shorter than 38 characters.

**H Home -** Sets the cursor at the beginning of the text.

**N End -** Sets the cursor at the end of the text.

**U Up -** Moves the cursor one page up.

**D Down -** Moves the cursor one page down.

**; EOL -** Moves the cursor to the end of the line.

**Clr Clear -** Deletes the whole text after check-back.

**B Start of block -** Defines the start of the block (line-wise) and cancels the old end of the block.

**E End of block -** Defines the end of the block. The length of the block may not be greater than 3K. The block is copied into a buffer, so it stays unchanged even if you edited the area of the block in the text. “Edit - Block defined” is shown on the bottom line, as long as a block is defined.

**C Copy -** Copies the block into the text at the location of the cursor.

**F Find -** Searches for a string. “?” is a wildcard for any character in the search text. Please use “?” instead of blanks.

**R Repeat -** Repeats the search with the same string.

**A 1-8 Activate -** Defines the number of the disk drive that is used for I.

**I Directory -** Shows the directory of the active drive. Press any key to return to the text.

**S Save -** Saves the whole text. If you haven't specified a filename yet, then you must type in a complete filename like D1:TEST.QIK. If a name was already used, the text appears in the bottom line and you can use it with or without editing it. To change the disk drive, you can simply type SHIFT 1-8. To print the text unformatted, just type “P:” as the filename. The text is sent to the printer.

**L Load -** Loads a text. The old text in RAM is deleted during this function. You can type in the filename of the text to load, or use the name of the file that was saved last.

**M Merge -** Appends a text at the end of the text in RAM. Don't forget that definitions of global variables must be copied to the start of the source text (see next section).

**O Overview -** Shows a brief help text for the control functions. Press any key to return to the text.

**Q Quit -** Leaves the editor after a check-back and returns to the shell. Type “Y” to leave the editor or anything else to return to the text.

You can see the editor has quite a lot of functions that you will not find in the BASIC editor.

# 6. The Compiler

Before telling you how to use the compiler, we would like to show you what the compiler really is doing during compilation.

If you don't know assembler you will probably not understand the following lines. But don't worry - you don't need to know assembler to program in QUICK. Just go to the next part of the manual.

## 6.1. The Compiler in Detail

The QUICK compiler's task is to translate QUICK ASCII source codes to machine language. The resulting programs shouldn't be too long (something that often happens in other languages) and the code should be fast. No compiler will ever meet both aims at a time, so we need to find a good compromise.

The QUICK compiler translates the text in 3 passes:

**Pass 1:** All variables are placed in the table of variables. The same is done with subroutines. The compiler also checks the structure of the program.

**Pass 2:** Now it's time for the real translation into machine code. There are 3 different types of mode:

**1. Assignments:** All transfers from one location into the other are directly translated into machine code.

Example:

```
A=B results in
   LDA B
   STA A
```

You can also use 16bit values or arrays.

**2. Macros:** Simple statements are translated into standard machine code segments. This is how a good macro assembler works:

Example:

```
SETCOL(N,F,H) results in
   LDX N
   LDA F
   ASL
   ASL
   ASL
   ASL
   ORA H
   STA 708,X
```

Because QUICK gets converted into machine code, it wouldn't be any faster to actually write it in machine code!

**3. Runtime subroutines.** Long and complicated statements call a subroutine in the runtime area of the QUICK program. The runtime is about 3K long, including all standard QUICK routines. It is linked to the program at the end of the compilation, so every QUICK program has a length of at least 3K.

Example:

```
   DIGI(G,A,E)    playing digitized sound
   MOUSE         follow the movements of an ST compatible mouse
```

**Pass 3:** Addresses are written into the right positions in the programs.

## 6.2. How to use the Compiler

After pressing “C” in the shell you enter the compiler. At first you have to type in the filename of the source code to be compiled. If you have saved a file in the editor before, the name of this file is the default in the compiler.

During compilation the line being compiled at the moment is shown on the screen, but this happens so fast that you won't be able to read it.

If the compilation succeeded, the end address of the program is shown and you'll see the following menu:

- **Another File:** Press “A” to compile another file.
- **Exit:** Press “E” to exit the compiler to the shell.
- **Save:** Press “S” to save the object file with the same name but with the extension “.OBJ”. The compiled QUICK programs are auto-start programs. That means they start automatically if you load them with the “L” load function from DOS.
- **Run:** Press “R” to start the program. To return to the shell later, you must press RESET.

Note: Use the upper case keys E,S,R,A only.

## 6.3. Error Codes

If an error appears during compilation, an error code is shown and the line in which the error is probably located is shown. You can also see the name of the subroutine in which this line is located.

Example: Error #05 in COLRS (C) (E)exit

This message tells you that error #5 (unknown statement) was used in the subroutine COLRS. If the error is not in a subroutine, MAIN will be shown. If the error occurs during the first pass, no exact location can be shown, and “???” appears instead of the subroutine name.

You may press “E” to exit the compiler, or “C” to continue the compilation. The final program probably won't work, but perhaps more errors are shown straight after.

And now to the most important question. What do QUICK source codes look like?

# 7. Programming in QUICK

Every source code has the same structure:

- Include block (possibly)
- Variable definition (possibly)
- Main program
- Subroutines (possibly)

## 7.1. Include

Libraries can be loaded during compilation by putting their name into the include block:

```
    INCLUDE
    (
        D1:GRAPH.LIB
        D8:MATH.LIB
    )
```

This structure (KEYWORD,[,FILENAME,]) is typical for QUICK. Every part has to be written on a separate line.

## 7.2. Definition of Variables

All global variables have to be declared in the variable definition block before the main program. Global variables are variables that are known throughout the whole program and they can be used everywhere.

In QUICK these variables must be declared so the compiler can put them into the table of variables. In QUICK there are 3 types of variables:

- **BYTEs:** They need one byte of RAM.
- **WORDs:** They need two bytes of memory.
- **ARRAYs:** These variables may be long up to 255 bytes. They can be used as an array of values or as strings.

The definition block looks like this:

```
TYPE
(
    NAME,NAME,...
    ...
)
```

The types are BYTE, WORD and ARRAY:

```
BYTE
(
    A1,F3,DA_S1
    I
)
WORD
(
    W1,WO
)
ARRAY
(
    VALU(10),TXT(40)
)
```

To define an ARRAY you must define the length of the array after its name.

## 7.3. Fixed Addresses

Normally all variables are stored in the table of variables. But there is also the facility to define the location of the variable. This can be very useful to write to OS variables:

```
BYTE
(
    COL1=708,COL2=709
)
```

Now you can write

```
    COL1=26
```

instead of

```
    POKE(708,26)
```

COL1=26 writes the value 26 into the variable COL1 which is located at the memory address 708. This is faster, shorter, and much better than using POKE or PEEK. By the way, SETCOL(0,1,10) does the same, but this takes much more time and space.

## 7.4. The Main Program

It begins with

```
    MAIN
```

and ends with

```
    ENDMAIN
```

The whole program is located between these two words.

## 7.5. Subroutines

They start with

```
    PROC Subroutine name
```

and normally end with

```
    ENDPROC
```

The first part of the subroutine is the declaration of local variables, which are divided into 3 groups: **IN, OUT and LOCAL.**

IN variables are given from the calling program to the called subroutine. OUT variables are given at the end of the subroutine from the subroutine to  the calling program.

Important: The sequence and the number of variables must be exactly the same in the subroutine's definition as at the calling of the subroutine.

Using LOCAL you can define variables that are only known inside the subroutine. So, it is possible to use variables in the subroutine that have the same name as variables in other subroutines (or as global ones). So you can use “I” as a variable in every subroutine without any interfering. Local variables are
very important for libraries, too. It is possible to use global variables in libraries, but not very sensible most of the time.

By the way, it is not possible to write recursive routines using standard variables.

BEGIN ends the definition block and starts the statement block of the subroutine.

Example:

```
PROC DEMO
IN
BYTE
(
    VAR1
)
OUT
BYTE
(
    VAR2
)
WORD
(
    WERT1
)
LOCAL
ARRAY
(
    TEXT(20)
)
BEGIN
    ...
ENDPROC
```

When calling this routine, one value must be given to the program and two variables must be taken back. The calling statement looks like this:

```
.DEMO(10,V1,W2)
```

or

```
.DEMO(V0,V1,W2)
```

Note the ‘.’. It is important.

In this example two values come back from the subroutine. If you use the same variable for both values

```
.DEMO(10,W2,W2)
```

then the variable' s value will be the value of the secondOUT variable. So, W2 has the same value as WERT1 in DEMO.

In DEMO there is a local ARRAY called TEXT which has a length of 20 characters (0 -19).

## 7. 6. Names of Variables

One thing is very important. In one part of the program there may never be two variables of the same degree (local/local or global/global) which have the same name. Also an IN variable should never have the same name as an OUT variable in the same subroutine.

Local variables have priority over global ones. That means, if a local variable with the right name exists this one will be used (and not the global one with the same name).

Finally you should never use a name that begins like a QUICK statement, e.g. MOUSEX as a name (as MOUSE is a QUICK statement). You will get strange error messages if you do it!

## 7.7. Static Variables

All variables in QUICK are static. This means that local variables don't loose their value at the end of a subroutine. If the routine is called again, their variables still have the same values as before.

## 7.8. Assignments

You can't assign a numerical term to variables in QUICK:

A=5-4*3  wrong!

On the right side of the sign of equality there may either be a number, a variable, or text:

A=5   right!
A=B  right!

Programming in “UNSIGNED MODE”, you can only use positive variables in QUICK V2.1. The following ranges are allowed:

BYTE 0 to 255
WORD 0 to 65535

If you switch to “SIGNED MODE” by using SIGN, you can use WORD variables in 2's complement. Now you can use the following range:

WORD -32768 to 32767

Whether a variable or number is interpreted without a sign or in 2's compliment doesn't depend on the type of variable itself. It is determined by the mode Quick is working in at the time. If UNSIGN is active, the variable is interpreted unsigned. If SIGN is active, it is interpreted in 2's compliment. This mode is the same for all variables. This is especially important for comparisons.

Here's an example:

The value 65535 is stored in the same way as the value -32768 in a WORD variable (like in assembler). Using UNSIGN this value will be treated for comparisons as 65535, but if you use SIGN before the comparison, the value is treated as -32768 (although the contents of the variable is the same in both cases if you just look at the 16 bits of the variable!).

The SIGN mode is also related to the PRINT statement. Normally WORDs and BYTEs are printed without signs. If you use SIGN, they will be interpreted in 2's complement and signs will be printed if the value is regarded as negative (which is the case when the highest bit of the BYTE or WORD is set).

In older versions of QUICK, BYTEs were also treated this way (not only WORDs). The PRINT statement still interprets signed BYTEs, so you should switch to the UNSIGNED mode whenever you don't need SIGN.

Don't forget: The (UN)SIGN mode has no meaning to the way QUICK is calculating. It is only necessary to select the right mode during comparisons or using PRINT statements (normally UNSIGN).

## 7.9. ARRAY Variables

ARRAY as string:

Text can be directly assigned to ARRAYs. If the text is longer than the dimension of the ARRAY, the text will be shortened to the right length:

TEXT="Hello"

The contents of one ARRAY can be copied to another ARRAY (the whole ARRAY will be copied - don't mix it up with the `C' language here!):

```
AREA=TEXT
```

You can also use an index here:

```
AREA="ABCDE"
AREA(3)=TEXT
Result: "ABCHello".
```

In this case the dimension of AREA is not checked during copying!

You cannot assign text in an indexed way:

```
AREA(3)="ABCD"  wrong!
```

The end of a string (concerning the PRINT statement and the routines of the string library) is marked with ZERO. So the dimension of the ARRAY should be at least one byte bigger than the length of the text.

ARRAY as CHR$ substitution:

In BASIC you can convert numbers into characters using the CHR$ function.
E.g.:

```
PRINT CHR$(125)
```

In QUICK you can do the same by assigning the number to an ARRAY and using the ARRAY as a string. Don't forget the ZERO at the end of the string:

```
AREA(0)=125
AREA(1)=0
PRINT(AREA)
```

ARRAY as a numeric one-dimensional array:

You can use ARRAYs as numeric arrays representing BYTE or WORD variables, thus using one byte or two byte variables:

AREA(0)=100 will write one byte containing 100 into AREA(0).

AREA(0)=!100 will write two bytes into AREA(0) and AREA(1) representing the value 100 in the same way as in a WORD variable.

You can also write AREA(2)=-!1000 for “negative” WORD values.

Don't forget to put a ‘!’ in front of all values that are to be placed in a  WORD. The compiler won't give an error statement if you write AREA(0)=60000, but it will not work the way you want it to!

## 7.10. The Statements

QUICK has about 60 statements, some of which you will know from BASIC, some from assembler, and some from `C'. Some you will not have seen on the ATARI at all. The compiler either converts the statements directly into machine code, or a subroutine of the RUNTIME area will be used. Such cases are marked with "x".

*** comment**

The '*' marks a comment. It should be at the beginning of the line. Sometimes you will also see the '*' at the right of a statement. We do not guarantee that this will work all the time (normally it does). (Some people (like me) also use the ';' on the right of a statement for short comments. ';' will never work at the beginning of a line!).

OPEN (NO,AUX1,AUX2,NAME)

Open channel number NO using the parameters AUX1 and AUX2 with the filename NAME. E.g.:

```
OPEN(1,4,0,"D1:TEXT.TXT")
```

Like in all other statements, the numerical parameters may only be BYTEs or WORDs. ARRAYs may only be used as string parameters, but not in a numerical way like AREA(0). So indexed use of ARRAYs is not allowed in statement or subroutine parameters:

```
OPEN (1,AREA(0),0,"P:")  wrong!

A=AREA(0)
OPEN(1,A,0,"P:")  right!
```

CLOSE(NO)

Closes channel number **NO**. Before opening a channel, you should be sure it is closed. It won' t do any harm closing an already closed channel.

BGET(NO,NUM,ADR)

Reads **NUM** bytes from channel **NO** at the address ADR. Before this, you should open channel **NO.**

BPUT(NO,NUM,ADR)

Writes **NUM** bytes from address **ADR** to channel **NO**.

INPUT (A)                    x

Allows the input of one number (A is a BYTE or WORD) or a text (A is ARRAY). You can' t input more than one thing at a time!

$XXXX operator

Value XXXX will be interpreted as hexadecimal. So you can write A=$D300 instead of A=54016. This is useful for many address values.

The sequence of all possible operators is "-!$" followed by a number.

PRINT (A1,A2,A3,...); or ?(A1,A2,A3,...);            x

Writes as many parameters to the screen as you want. In the brackets you can put in many parameters separated by ",". You can use BYTEs, WORDs and ARRAYs:

```
PRINT(A,B,10,"Hello world",AREA)
```

After numbers, a space will be inserted automatically. At the end you can add ";" so no NEWLINE is written. The next PRINT will begin at the position the other one ended, then:

```
PRINT("Hello, ");
PRINT("world")
```

produces **Hello, world**

LPT(A,B,...);                    x

Writes to channel 5 the same way PRINT writes to the screen. This statement can be used, for example, to send data to the printer or to a file in an ASCII formatted way. You must OPEN the channel before:

```
OPEN(5,8,0,"P:")
LPT("This is printed by the printer")
```

SIGN

Signs are now used for PRINT and comparisons for WORD variables.

UNSIGN

All variables are interpreted as positive numbers.

POS(X,Y)

Puts the cursor to the location X,Y on the screen.

LOCATE (VAL)                    x

Reads the value of the screen location of the cursor into VAL.

COLOR(A)

Selects colour register 0 to 15 for PLOT and DRAW.

PLOT(X,Y)                    x

Puts a pixel of the chosen colour to the position X,Y.

DRAW (X,Y)                    x

Draws a line between the position of the cursor and X,Y.

PLAYER(Z,I,L,S)

Copies **L** bytes from address **S** to page **Z**, using **I** as an index being added to **Z**. This is very useful for copying player data to the right position in the player area. Here, index **I** can be used as "Y position" of the player. **I** may range from 0 to 255. E.g.:

```
PLAYER(128,20,8,8000)
```

copies 8 bytes from address 8000 to address 128*256+20 (=32788).

CLR(P,NUM)

Deletes NUM*256 bytes beginning at page **P**.

CUT(X1,Y1,X2,Y2,ADR)                x

Copies a rectangle with the corner positions (X1,Y1) and (X2,Y2) into memory at address ADR. It is stored there in a special format. This statement works only in Graphics 8 (and also in 7 or 15, but here you must multiply the X-values by 2 and add 1). So QUICK is the first language with a built in Blitter.

PASTE(MODE,X1,Y1,ADR) x

Copies the formerly cut data from memory to the screen. ADR is the position where CUT put the data before, X1 and Y1 is the new location to put the graphics to. MODE defines the way the data is copied to the screen. If MODE=0, then it is simply copied. If MODE=1 it's copied in OR-mode (only set
points will be copied). This mode is probably only sensible in Graphics 8.

SETCOL(NO,C,B)

Sets colour register **NO** to colour **C** and brightness **B**.

SOUND(C,H,D,V)

Activates sound channel **C** with volume **V** and distortion **D** at frequency **H**. Note that the distortion has a range between 0 and 7. Just divide the value you use in BASIC by two to get the right one for QUICK.

DIGI(T,S,E) x

Plays digitized sounds. The data is played from address **S** to address **E**. Only the High byte of the address is important here. The speed **T** can range from 1 to 255 (fast to slow) or it can be 0. Normally the screen is switched off during playing, but if T is zero the sound is synchronised with the VBI, so has one specific speed (rather slow).

The data must be in a special format as defined in ATARI magazine 1/1989 and 5/1989 by PPP Germany (there you can also find how to build a sound sampler for the XL/XE computers). The format looks like this: Two 4-bit sound data are put into one byte. The first data is put in the high nibble, the second in the low nibble of the byte. The third sound data is put in the high nibble of the second byte and so on. Samples from Parrot, Replay, Digi-Studio, and Antic Sampling Processor are fine.

MOUSE x

Writes the actual position of a mouse into **MX** and **MY**. **MX** and **MY** are variables with fixed addresses. **MX** is at address 178 and **MY** at 179. You must define them with:

```
BYTE
(
MX=178,MY=179
)
```

before using them.

The mouse you should use is an ATARI ST compatible mouse in joystick port 2.

To follow the movements of a mouse, simply call MOUSE in a loop, where you don't do many other things. MOUSE will follow the mouse for some 50 milliseconds all the time. So it is easy to follow fast movements, too. By writing into **MX** and **MY**, you can also “set the mouse” to a specific position. Use the PLAYER statement to display a mouse arrow.

SYNC(X)

Puts X synchronization statements into the machine code. X must be a number, not a variable.

This statement is useful for Display List Interrupts. If you want, for example, to change a colour at the end of a line without flickering, you should write SYNC(1) before changing the colour. This will make the CPU wait until the electron beam of the TV reaches the end of the line, so the changing of the colour is not seen somewhere in the middle of the TV line.

It is also useful to wait for 16 lines to be shown (SYNC(16)) to change the colour after 16 lines again, without using a second DLI.

CALL(ACCU,X,Y,ADR)

Calls a machine code subroutine at address ADR. Note that ADR must be a number, not a variable. (If ADR is a variable CALL will jump to the location of the variable, not to where the contents are pointing!). Before calling, X is copied into the X register of the CPU, Y to the Y register and ACCU to the Accumulator. If you want to use variable calling, look in appendix A.

POKE(ADR,B)

Writes the contents of B (8 Bit) into the address ADR is pointing at.

PEEK(ADR,B)

Writes the contents of the address ADR is pointing at into the variable B (8 Bit).

DPOKE, DPEEK

The same as above, but for 16 Bit.

DATA(ADR)

```
DATA(123)
(
1,4,876,4563,34,...
)
```

Writes data beginning at address 123 into memory. The address may be a number or an ARRAY. Thus, the data will be copied into the ARRAY:

```
DATA(STR)
(
$1B,120,0
)
```

This will copy the data 27,120,0 into the ARRAY STR. This is very good for escape codes for the printer!

BMOVE(S,D,L)                    x

Copies the memory from address **S** with a length of **L** to the address **D** to **D+L-1**. It is possible to copy overlapping areas.

FMOVE(S,D,L)                    x

Does the same as BMOVE, but L must be between 0 and 255. The areas should not overlap. FMOVE is faster than BMOVE.

INLINE

Writes the data between the brackets directly into the compiled program and will run into this area, so the data should be a real machine code program:

```
INLINE
(
169,45,141,A1
)
```

The data cannot only be numbers, but also variables. In this case their address will be used as a value. Normally a two byte value will result as a variable's address. But if the address of the variable is between 0 and 255 (in page zero), only one byte is inserted. If you know how to program in assembler, you'll know why we do it that way. If you don't know how to program in machine code, don't think about this statement any more. The INLINE statement is good for extremely fast DLIs and other short but fast program parts.

REGX(Z), REGY(Z), REGA(Z), REGP(Z)

Copies the X or Y register or the Accu or the status register into the variable **Z**. If **Z** is a WORD only the low byte will be changed! **REGP** changes the Accu itself, so you must use the following order to avoid incorrect results:

```
REGA(VAR2)
REGP(VAR1)   right!
```

```
REGP(VAR1)
REGA(VAR2)   wrong!
```

PADR(SUBROUTINENAME)

This statement copies the address of the routine SUBROUTINENAME into the registers 208,209 ($D0,$D1). Simply use a WORD with a fixed address to get the address of a subroutine:

```
WORD
(
ADR=208
)
...
PADR(SUBROUTINENAME)
?("The address of SOUBROUTINENAME is",ADR)
```

VADR(VARIABLENAME)

Copies the address of the variable VARIABLENAME into 208,209.

/VARIABLENAME

'/' is the high-word operator. The result is a BYTE containing the high-byte of the WORD variable's contents.

ADD(A,B,C)

Calculates **C=A+B**. WORDs and BYTEs may be mixed. But, there is no message if an overflow appears (unlike in BASIC).

A+

Increases the BYTE variable A by one.

SUB(A,B,C)

Calculates **C=A-B**

A-

Decreases the BYTE variable A by one.

MULT(A,B,C)                    x

Calculates **C=A*B**

DIV(A,B,C)                    x

Calculates **C=A/B**

AND(A,B,C)

Calculates a bitwise AND between A and B. The bits of the two values are calculated one by one (this differs very much from BASIC!). A bit in C will be set if the same bit is set in A and B:

```
 A=6        (bin: 00000110)
 B=3        (bin: 00000011)
 AND(A,B,C) (bin: 00000010) in C
```

OR(A,B,C)

Calculates A OR B. A bit in C will be set if it is set in A or B.

EOR(A,B,C)

Calculates A EOR B. A bit in C will be set if is not the same in A and B.

ASLB(A)

Shifts the value A (8 bit) 1 bit to the left. Numerically this means the value is doubled.

```
 A=5      (bin: 00000101)
 ASLB(A)  (bin: 00001010) = 10
```

ASLW(A)

The same for 16 bit values (e.g. WORDs).

ASRB(A)

Shifts the value A one bit to the right (the highest bit is unchanged, so the sign of the value will be unchanged). But this is only really correct if this variable will be used in SIGN mode -- so this statement will not be used very often, as BYTEs are always unsigned). So the value is divided by 2 very fast.

ASRW(A)

The same for 16 Bit. For WORDs the SIGN mode is interesting, so this statement is important.

LSRB(A)

Shifts the 8 bit values completely one bit to the right. The highest bit will be filled with zero. This statement is correct for all variables in UNSIGNED mode. It is also a fast division by 2.

LSRW(A)
The same for 16 Bit.

## 7.11. Control Structures

### 7.11.1 Jumps

You can define labels and jump to their position. A label is a dash, followed by a number between 0 and 384:

```
 -10
```

To jump to a label write:

```
 JUMP(10)
```

Don't use jumps too often. There are better ways for loops and other things like that.

### 7.11.2 Loops

There are two possibilities for doing loops:

```
 REPEAT
   ...
 UNTIL comparison
```

and

```
 WHILE comparison
   ...
 WEND
```

The code at **...** will be executed as long as the comparison is true.

### 7.11.3 Comparisons

A comparison is a statement of type value operator value. Available operators are:

```
=, >, <, >=, <=, <>
```

So you may write:

```
 A>C
 A=M
 G<>100
```

But not:

```
 FELD="ABCD"
 A=B OR C=D
```

That means you can't use ARRAYs for comparisons and you do not have a boolean OR or AND in QUICK.

WHILE WEND is a rejective loop. At the beginning the condition is checked. If it is false, QUICK will jump to WEND immediately. In the other case, the block will be repeated until the condition becomes false.

REPEAT UNTIL checks the condition at the end of the loop, so the loop will be run through at least once.

Don't forget that you have to change the condition yourself (e.g. increase a loop variable -- unlike the FOR NEXT in BASIC). Example:

```
 I=0
 WHILE I<10
   ?("I is less than 10")
   I+
WEND
```

### 7.11.4 Conditional Execution

A very important control structure is:

```
 IF comparison
   ...
 (ELSE
   ...)
 ENDIF
```

Example:

```
 IF A<10
   ?("A is less than 10")
 ELSE
   ?("A is greater or equal to 10")
 ENDIF
```

## 7.12. Interrupts

In QUICK any subroutine can be used as a Display List Interrupt or Vertical Blank Interrupt. Instead of beginning the routine with PROC use INTER and instead of ENDPROC use ENDDLI or ENDVBI.

### 7.12.1 VBI

Programming a VBI is very easy in QUICK. Just write VBI(VNAME) to start the routine VNAME as a VBI. So this routine will be called by the operating system 50 times (PAL systems, e.g. in Europe) or 60 times (NTSC systems, e.g. in the USA) a second at the end of displaying the screen.

### 7.12.2. DLI

A DLI is installed by using DLI(DNAME). Unlike the VBI you must store the CPU registers for later use of the main program yourself. So at the beginning of the DLI routine you must write PUSH and at the end PULL, so the registers are saved from changes during the interrupt.

Here is an example:

```
INTER TEST
      LOCAL
      WORD
      (
        SP, OP=130
      )
      BEGIN
      IPUSH
      ZPUSH
      SP=OP
      ..........
      OP=SP
      IPULL
      ZPULL
      ENDVBI
```

where TEST can be replaced by the name of your VBI and ........... should be replaced by the code you want to be used as the VBI, this of course can be more than one line!

For DLIs you must also include the command PUSH immediately before IPUSH, and then PULL immediately before IPULL but after OP=SP, and change ENDVBI to ENDDLI. Some commands may not be necessary; it depends on exactly which commands you use inside the DLI.

### 7.12.3. Saving QUICK's Variables

In the above way, only the CPU registers are saved for the main program. But if you use QUICK routines in the interrupt which are marked by a “x”, you may also change QUICK's internal variables. In this case, you must save QUICK's variables, too:

```
 IPUSH
 ZPUSH
 ...
 IPULL
 ZPULL
```

IPUSH/PULL is necessary if you use comparisons (with WORDs) or x-routines.

ZPUSH/PULL is necessary if you use PEEK or POKE or *-routines in your interrupt.

If you use PEEK or POKE in the VBI or DLI, you have to save the registers 130,131 yourself, as they are changed by PEEK and POKE and are not restored automatically at the end of the interrupt. Use this command block to save the registers:

```
LOCAL
WORD
(
ORGPEEK=130
SAVEPEEK
)
...
BEGIN
 SAVEPEEK=ORGPEEK
 ...
 ORGPEEK=SAVEPEEK
ENDVBI
```

### 7.12.4. Local Variables

You may define local variables in the interrupt routines. But, you can't use IN or OUT variables because the interrupt is not called by QUICK but by the OS. So, it's impossible to give values to the routine.

### 7.12.5. Spare Time

Don't forget that a DLI may only take up to 10 milliseconds of time. A VBI may be longer but not longer than 1/50 or 1/60 of a second -- normally, at least.

### 7.12.6. Some Basics

An interrupt is a short machine code routine that is called by the OS in short, periodic intervals:

- VBI: A VBI is done whenever the electron beam of the monitor reaches the bottom (50 or 60 times a second). Now a short program can do interesting things like moving a player or mouse pointer, playing sounds and many other things. It is not sensible to write a VBI that takes more than 1/50 or 1/60 of a second.

- DLI: A DLI is called every time the graphics chip reads a DLI command in the Display List. This is the case when a value bigger than 127 is placed in the DPL. Normally a DLI is used to change the screen colours in an exactly defined line. Normally DLIs are very short (only a few statements). So if you write DLI(DNAME) nothing will happen until you change the DPL in a specific way. Take a look in a good book!

## 7.13. Libraries

QUICK is a language that is easy to extend. This is achieved by “libraries”. A library is a source code consisting of many subroutines. During compilation the file can be loaded by the compiler. A library has a simple structure. You can write many PROC/ENDPROC (OR INTER/END...) blocks in the file.
There can't be a MAIN block, and there can't be any definitions of global variables.

To use the routines of a specific library, you must put its filename in the include block at the very beginning of the source code of the main file:

```
INCLUDE
(
D8:STR.LIB
)
```

Now you can use all the routines with normal ‘.’ calls. Please note that some routines could change the UNSIGN/SIGN mode.

Four libraries are supplied on the system disk. Many others are available on the Support Disks.

### 7.13.1 The graphics library: GRAPH.LIB

This library contains a lot of useful graphical routines:

.GRAPHICS(GR)

Opens a graphics screen just like the BASIC GRAPHICS() statement. The graphics mode may be 0 to 15. Adding 16 will prevent a text window being opened and adding 32 will leave the screen uncleared.

.FRAME(X1,Y1,X2,Y2)

Draws a rectangular frame between the corner positions X1,Y1 and X2,Y2.

.BOX(X1,Y1,X2,Y2)

Draws a solid rectangle.

.CIRCLE(X1,Y1,R)

Draws a circle around X,Y with radius R. This routine works from Graphics 4 to 15.

.DISC(X1,Y1,R)

Draws a solid disc.

.FILL(X,Y,X1,Y1,X2,Y2)

Fills a single coloured screen area beginning with X,Y. With X1,Y1 - X2,Y2 the area to be filled can be defined.

### 7.13.2. The String Library: STR.LIB

This library offers a lot of routines to manipulate ARRAYs of 1 to 254 bytes in length like strings (in C or TurboBASIC).

This library is using something like “global memory” to minimize the memory consumption. Instead of using normal IN and OUT variables (ARRAYs) it uses IN and OUT variables with fixed addresses in the area of $A800 to $ABFF. So if you use this library don't use this area. If it is necessary you could change these (in all routines of the library!!!).

```
.STRCMP(STRIN1,STRIN2,RES)
```

Compares two ARRAYs and has the results:

```
1 if S1=S2
2 if STRIN1>STRIN2
3 if STRIN1<STRIN2
```

During comparison upper and lower case characters are treated with different values so “ABC” is not the same as “abc”.

```
.STRCHR(STRIN,CHR,CHRPOS)
```

Searches for the character CHR in STRIN and returns 0 to 254 as position of CHR or 255 if the character was not found.

```
.STRCAT(STRIN1,STRIN2,STROUT)
```

Puts STRIN2 at the end of STRIN1 and returns this result in STROUT. The length of STRIN1 and STRIN2 together should not be greater than 254.

```
.STRLEN(STR,RES)
```

Calculates the length of STR. (The end of strings are marked by a ZERO.)

```
.STRUPR(STRIN,STROUT)
```

Changes lower case characters to upper case.

```
.STRGRP(STRIN,STROUT)
```

Changes upper case to graphic characters (e.g.: ‘T’ will be the small circle-sign).

```
.STRLWR(STRIN,STROUT)
```

Changes upper case to lower case.

```
.STRICMP(STR1IN,STR2IN,RES)
```

Compares STR1IN with STR2IN, but without distinguishing between upper and lower case.

```
.STRCUT(STRIN,SSTA,SEND,PARTSTR)
```

Returns in PARTSTR the part of STRIN between STRIN[SSTA] and STRIN[SEND].

```
.STRSET(CHR,NUM,STROUT)
```

Fills STROUT with NUM characters of CHR.

```
.STRINV(STRIN,STROUT)
```

Adds 128 to every character in STRIN.

.STRSTR(STRIN1,STRIN2,SPOS)

Searches for STRIN2 in STRIN1 and returns 0 to 254 if STRIN2 was found or 255 if it was not found.

.STRCPY(STRIN1,STRIN2,S,E,SPOS,STROUT)

Copies STRIN2[S] to STRIN2[E] to STRIN1 at the position SPOS and returns the resulting string in STROUT.

.STRWRT(STRIN,ADR)

Copies the string into memory at ADR. This is useful for generating 2-dimensional string ARRAYs.

.STRRED(ADR,STROUT)

Reads the string STROUT from memory at ADR.

The demo program STRDEMO.QIK shows how to use the routines of STR.LIB. It's very easy -- compared with ATARI BASIC!

### 7.3.13. The math library: MATH.LIB

This library offers features that allow the use of floating point variables the XL/XE OS offers. In QUICK a floating point variable is defined as an ARRAY of 6 bytes. Byte 0 contains the exponent and the sign, the other 5 bytes represent 10 digits in BCD format. To use the library for floating point computations you don't need to now the exact format of the OS internal variables (just like in BASIC).

As QUICK has no built in floating point abilities, every long calculation must be reduced to the following fundamental steps.

.IFP(WORD,FLOAT)

Converts an integer UNSIGNED WORD value into FP format and returns the result in the ARRAY FLOAT.

.FPI(FLOAT,WORD)

Converts a FP variable into a WORD variable. FP should not be greater than 65535.0 or negative as the word variable can't be this.

.AFP(ASC,FLOAT)

Converts an ASCII-string (any ARRAY with digits) into a floating point variable.

.DFP(D1,D2,D3,D4,D5,D6,FLOAT)

Builds the FP variable of the 6 bytes D1 to D6. This is useful if you know the internal format of a FP number. You can give a FP variable any value without using AFP (which would be slower).

.FASC(FLOAT,ASC)

Converts a FP value into a string. Often the string is not formatted very well. To print a FP variable, use FPRT.

.FADD(FIN1,FIN2,FOUT)

Adds the FP variables FIN1 and FIN2 and returns the sum in FOUT.

.FSUB(FIN1,FIN2,FOUT)

Subtracts FIN2 from FIN1.

.FMUL(FIN1,FIN2,FOUT)

Multiplies FIN1 by FIN2.

.FDIV(FIN1,FIN2,FOUT)

Divides FIN1 by FIN1.

.FCMP(FLOAT,BYTE)

Compares a FP variable with zero. It returns 0 if FLOAT is negative, 1 if it is zero and 2 if it is greater than zero.

.FPRT(FLOAT)

Prints a FP value in a formatted way to the screen. To copy one floating point array (F1) to another (F2) just write F2=F1. All 6 bytes will be copied this way.

### 7.13.4 The numeric library: NUMERIC.LIB

This library contains a lot of numerical routines:

.EXP(FIN,FOUT)

Calculates e to the power FIN and returns the result in FOUT.

.EXP10(FIN,FOUT)

Calculates 10 to the power FIN.

.LOG(FIN,FOUT)

Calculates ln(FIN).

.LOG1O(FIN.FOU

Calculates log(FIN).

.SQR(FIN,FOUT)

Calculates the square root of FIN.

.SIN(FIN,FOUT)

Calculates the sine of FIN (in radians).

.COS(FIN,FOUT)

Calculates the cosine.

.ATN(FIN,FOUT)

Calculates the arctangent of FIN.

.ABS(FIN,FOUT)

Calculates the absolute value of FIN.

### 7.13.5. Programming Your Own Libraries

You don't need to consider many special things when programming your own libraries. As all libraries are mentioned in the INCLUDE block at the beginning of the source code, but being linked to the program at the end of the source code in reality, it is not possible to define global variables in a library. You can only define local variables in the routines.

You may use global variables that are defined by the main program, but this could be dangerous, and the library will depend on this special main program forever.

You also shouldn't use too many JUMPs/LABELS in libraries. As JUMPs are always global, duplicate JUMPs may occur and will interfere badly. Don't use the same labels in your libraries as in our libraries on the disk.

You should never change the standard libraries! Copy them with a new name on the disk if you want to enhance one of our routines. Otherwise, it is possible that sources you get from other QUICK programmers will not work on your compiler.

In comparison with PASCAL, QUICK always links the whole library to the program no matter if one or all routines of the library are used in the program, so compiled programs will get longer. Because of this it would be a good idea to copy a single routine from the library to the main code, if no other routines from this library are used. This will spare memory – if you really need it.

### 7.13.6. Demo Programs

You will find some demo source codes on the system disk. There are a lot of comments in these files. Load them into the editor to study them. We also show how to use the libraries there.

The program LIST.OBJ is a program that prints QIK source codes in a nicely formatted way on EPSON compatible printers. Just compile LIST.QIK, or run the compiled program from the DOS shell. Then you will be asked which .QIK to print. You can type in the name, and also the date, time and printing quality.

## 8. Error Messages

If something goes wrong during compilation, the compiler will present different error messages. These codes are sometimes not very exact. It is possible that the position is not located in the right line (that's a problem of most compilers).

Like in most compiler based languages, the running program will not make any error messages any more. If something goes wrong during runtime, the computer may crash. So, don't trust RAMDISKs too much!

Here are the error codes:

01 '(' is missing

02 name of PROC is missing

03 MAIN doubly defined
  There may only be one Main program in a source code

04 MAIN is missing, or PROC before MAIN
  Always follow the sequence of MAIN, then PROC

05 unknown statement

06 invalid value

07 unknown variable
  The variable was not defined

08 invalid assignment
  Wrong type of variable used, e.g. text was assigned to a BYTE

09 Value too big

0A Dimension of ARRAY is missing
  '(x)' on the right of the ARRAY name at the definition is missing

0B Dimension of ARRAY too big
  The dimension must be smaller than 255

0C invalid value used as index

0D unknown subroutine
  A subroutine is called that was never defined. Possibly the INCLUDE statement is missing.

0E too few parameters at subroutine call
  There must be the same number of parameters passed to a subroutine as in its definition

0F internal error
  Heck! How did you do this?

11 ARRAY with index not allowed here
  Only BYTE or WORD may be used in some statements

12 No index allowed
  Sometimes only ARRAYs without index are possible

13 Error at INCLUDE
  A library cannot be loaded

14 Parameter is missing
  wrong statement syntax

15 ')' is missing

16 only BYTE allowed here

17 '(' is missing

18 only BYTE or WORD allowed here

19 Operator is missing
  The comparison doesn't have the right syntax

1A internal stack overflow
  Don't use so many IF's and LOOPs in a hierarchical order

1B wrong hierarchical order of IF, REPEAT or WHILE

1C IF, REPEAT or WHILE missing
  ENDIF, UNTIL or WEND found without the right start

1D Text can not be compiled
  You probably typed in the wrong filename for the compiler to compile

1F Number of label invalid
  Value is too big (greater than 384) or negative

20 compiled program too long
  split it in two parts and load one during runtime by the other...

21 Too many variables

22 Too many subroutines

23 Too many subroutine calls or JUMPs

24 Too many INCLUDE files

25 BEGIN is missing
  BEGIN must be placed after PROC somewhere, before ENDPROC

26 wrong type of variable
  A wrong variable is used in a statement

27 ENDIF, UNTIL or WEND is missing
  This error will always appear only at the end of a program part (e.g.
  after ENDMAIN if MAIN is missing). Check the hierarchy of the source code

28 RUNTIME.OBJ cannot be loaded
  The compiler did not find it on D1: or D8:

29 Label is missing
  There is a JUMP to a label that does not exist

# 9. Appendix A -- Hints

Now it's time for you to start programming in QUICK. Here is some additional information to get the most out of QUICK.

## 9.1. Optimization

Like in every programming language, there are many different ways to solve a special programming problem. Sometimes it is better not to use a complex QUICK statement, but to go back to the roots.

If you would like to change the colours in an interrupt for example, you should not use SETCOL, because here the different parameters must be multiplied by QUICK to find out the right value to be written in the colour register. It is much faster to assign a constant value to this register by using a variable with a fixed address:

A little bit slow:

```
BYTE
(
H
)
...
REPEAT
H+
SETCOL(0,0,H)
UNTIL H=15
```

Extra fast:

```
BYTE
(
H=708
)
...
REPEAT
H+
UNTIL H=15
```

You see, the INC statement (+) is used, as it is still faster than ADD().

Sometimes it is necessary to give a lot of variables the same value, e.g.:

```
  A=0
  B=0
  C=0
```

To make this super fast, use the REGA statement:

```
  A=0
  REGA(B)
  REGA(C)
```

But don't forget: QUICK is fast -- without special tricks!

## 9.2. Logical Statements

In BASIC you can write:

```
  IF A=1 AND B=2 ...
```

Such double comparisons are not possible in QUICK (and moreover there is no logical AND in QUICK). So you must split this statement in to parts:

```
IF A=1
  IF B=2
    ...
  ENDIF
ENDIF
```

## 9.3. Memory Map

Sometimes it is important to know where there is free memory, e.g. for players, character sets, graphics and so on.

The following memory map is correct for the moment after compilation (and after starting the compiled program):

| | |
|---|---|
| 1C70 - 1EFF | SHELL |
| 1F00 - 40FF | COMPILER |
| 4100 - 4FFF | RUNTIME |
| 5000 - XXXX | PROGRAM |
| XXXX - B000 | FREE |
| B000 - BFFF | VARIABLES |
| C000 - C7FF | EDITOR PART1 |
| C800 - DFFF | OS |
| E000 - FFFF | EDITOR PART2 |

The space between 1536 and 1792 is used for temporary storage of the  filename during the change from the compiler to the editor. But you may use this area, if you want.

Theoretically there is free memory from the end of the program (XXXX) to AFFF. But please note that the screen memory is below AFFF. This means you lose about 1 to 7K for the screen.

If you only use this free RAM you will return (after pressing RESET) to the shell and the compiler.

If your program needs more RAM, you can use the RAM where the compiler is stored normally (1F00 -- 40FF). But, the compiler is now destroyed. So, you either must boot again, or at least you should go to DUP and reload AUTORUN.SYS.

The area of the RUNTIME should never be over-written by a program!

There is an easy way to find out the address of the first free byte after the program. First it is shown after the compilation. It is also possible to find out this value during runtime of the program. The value is available in $4FFE, $4FFF. Just use the following block:

```
WORD
(
 FREE=20478
)
 ...
 ?("First free byte ",FREE)
```

## 9.4. Don't Forget the Compiler

The compiler sets some system variables to special values: the left skip is changed, the colours are black and white, and so on. If you start your program with Run from the compiler, these changes are active. But if you start your program from DUP or via booting, some things may look very different!

## 9.5. Assembler in QUICK -- Only for Specialists!

Using the INLINE statement it is possible to write machine code directly into the program code. All data in the INLINE block is immediately written to the program. This data must be real machine code, not just any data. On the other hand you could jump over this area with JUMP, but then you could use DATA(), too.

INLINE may be good for time critical routines (like interrupts).

For example you could write your own IF. But, we don't think that this would be much faster.

Normal QUICK:

```
BYTE
(
  VALUE
)
...
IF VALUE=5
  ?("IT'S FIVE")
ENDIF
```

With INLINE:

```
BYTE
(
  VALUE
  LDA=173, BEQ=240, CMP=224
)
...
INLINE
(
  LDA,VALUE,CMP,5,BEQ,3
)
JUMP(1);     Jumps is 3 byte long
?("IT'S FIVE")
-1
```

As already stated, the address of a variable is inserted at its position in the INLINE statement. If the address is lower than 256, only 1 byte is inserted, else 2 bytes are out in the code. So it is possible to define assembler statements. If you use the fixed address 224 for the CMP variable, the value 224 will be inserted in the INLINE block. This is the value for the machine code for CMP.

You should never assign a value to such a variable!

## 9.6. QUICK and DOS

In principle QUICK should work with any DOS and DUP. But as the shell is located at the low address of $1C70, the DOS must be a very short one. DOS 2.5 is fine for QUICK.

QUICK may also work with other DOS systems. To spare some memory don't use too many drives and too many open files at the same time.

## 9.7. Version 1.x

To compile older source codes from QUICK Version 1.x, you should write the statement OLD in the first line of the source code. Now QUICK Version 2.x will react like the old one concerning SIGNED BYTEs. But programs compiled with the OLD option will be slower and longer than normal ones.

## 9.8. CALL with Variable

To jump to the address that is given by the contents of a WORD variable SP, use the following lines, which will call a machine code routine at $8000.

```
WORD
(
  PDR=208, SP
)
...
SP=$8000
PADR(JSR)
ADD(PDR,1,PDR)
DPOKE(PDR,SP)
.JSR

...

PROC JSR
BEGIN
INLINE
(
  32,0,0
)
ENDPROC
```

## 9.9 Calling DUP

To jump from a QUICK program to DUP, write:

```
INLINE
(
  108,10,0
)
```

## 10. Summary of Commands

Shell:

| | |
|---|---|
| E | Call editor |
| C | Call compiler |
| D | Call DOS |
| Control-Q | Cold boot |

Editor:

| | |
|---|---|
| TAB | Insert 2 blanks |
| ESC | Text/Graphic characters |
| CTRL-Delete | Delete char |
| Clear | Delete text |
| ; | Goto EOL |
| H | Home |
| N | End |
| U | Page up |
| D | Page down |
| X | Delete line |
| V | Connect lines |
| B | Block start |
| E | Block end |
| C | Insert block |
| F | Find |
| R | Repeat find |
| L | Load text |
| M | Append text |
| S | Save text |
| I | Show directory |
| A&1-8 | Set drive no. |
| O | Help |
| Q | Quit |

Compiler:

| | |
|---|---|
| R | Run program |
| S | Save program |
| A | Compile another program |
| E | Call shell |

Quick Commands:

Blocks:

| | |
|---|---|
| INCLUDE | Load libraries |
| BYTE,WORD,ARRAY | Variable declaration |
| MAIN,ENDMAIN | Main program block |
| PROC,ENDPROC | Subroutine program block |
| INTER,ENDVBI | VBI-Block |
| INTER,ENDDLI | DLI-Block |
| INLINE | Machine Code block |
| DATA(adr) | Data Block |

I/O-Commands:

| | |
|---|---|
| OPEN(k,a1,a2,f) | Open channel |
| CLOSE(k) | Close channel |
| BGET(k,anz,adr) | Load data |
| BPUT(k,anz,adr) | Save data |
| INPUT(a) | Read data from "E:" |
| PRINT(a,b,...); | Write data to "E:" & "S: |
| LPT(a,b,...); | Write data to channel 5 |

Graphic commands:

| | |
|---|---|
| POS(x,y) | Cursor to x,y |
| LOCATE(w) | Value below cursor |
| COLOR(c) | Choose colour |
| SETCOL(n,f,h) | Set colours |
| PLOT(x,y) | Set pixel |
| DRAW(x,y) | Draw line to x,y |
| CUT(x1,x2,y1,y2,adr) | Cut a part from screen |
| PASTE(m,x,y,adr) | Paste back to screen |
| PLAYER(z,i,l,q) | Copy player data |
| MOUSE | Handle ST-Mouse |

Sound commands:

| | |
|---|---|
| SOUND(k,h,v,l) | Switch sound on |
| DIGI(g,a,e) | Play digital sample |

Memory related commands:

| | |
|---|---|
| POKE(a,b) | Write a BYTE to RAM |
| PEEK(a,b) | Read a BYTE from RAM |
| DPOKE,DPEEK | Same for WORD |
| BMOVE(q,z,l) | Copy memory |
| FMOVE(q,z,l) | Copy memory faster |
| CLR(p,anz) | Clear pages |

Machine code related commands:

| | |
|---|---|
| CALL(a,x,y,adr) | Call a machine code prg. |
| REGA(a),REGX(a) REGY(a),REGP(a) | Read CPU register |

Arithmetic commands:

| | |
|---|---|
| ADD(a,b,c) | C=A+B |
| SUB(a,b,c) | C=A-B |
| a+A=A+1 a-A=A-1 | (a is BYTE) |
| MULT(a,b,c) | C=A*B |
| DIV(a,b,c) | C=A/B |
| AND(a,b,c) | C=A AND B |
| OR(),EOR() | like AND |
| ASLW(a),ASLB(a) | 1 Bit left |
| LSRW(a),LSRB(a) | 1 Bit right |
| ASRW(a),ASRB(a) | "   with sign |

Control structures:

**IF <Comparison>  (ELSE)  ENDIF**

**WHILE <Comparison> WEND**

**REPEAT UNTIL <Comparison>**

**JUMP(x)**

Misc:

| | |
|---|---|
| SYNC(n) | Line end synchronization |
| PADR(n),VADR(n) | Get subprg., variable address |
| DLI(n) | DLI on |
| VBI(n) | VBI on |
| .n | Call subroutine "n" |
| PUSH,PULL<br>IPUSH,IPULL<br>ZPUSH,ZPULL | Save registers |
| SIGN | Use signed numbers |
| UNSIGN | Use unsigned numbers |
| OLD | V1.6 Mode |

## 11. Credits and Copyright

The Quick programming language was written by Harald Schönfeld and Andreas Binner between 1989 and 1990. It is published in Germany by PPP. It is published in the English language format by Dean Garraghty. Quick is copyright 1990, Raindorf Soft. Copying this software is an infringement of copyright. Action may be taken by any person(s) known to be illegally copying this software.

This manual was translated by Harald Schönfeld. It was edited and typeset by Dean Garraghty. The same copyright applies to this manual as to the programs.

Neither PPP, or Dean Garraghty, can be held responsible for any errors which may occur in the use of this software. We do not claim it is useful for all applications.

Dean Garraghty,
February 1993,
International Version 1, Release 3 (Jul 2002).