DJ TOOLKIT

v1.15

USER GUIDE

by Norman Dunbar

Published by DJC

Contents

DJTOOLKIT

WRITTEN BY NORMAN DUNBAR, 1993-94

This has been produced at the suggestion of Dilwyn Jones in an effort to provide QLiberator users with some of the file & memory handling utilities, direct file access (in internal format) & positioning commands that are found in the Turbo Toolkit and Toolkit 2 etc. In addition, there are a few routines not (yet) found in any other toolkit.

All of the procedures and functions in this toolkit can be compiled by QLiberator, but one of the functions, DEV_NAME, cannot be compiled by Turbo or Supercharge as it modifies its parameter as well as returning a string.

This toolkit may be supplied as part of a commercial or shareware or public domain program which has been QLiberator compiled. It should be supplied linked to the object program, not loaded by a BOOT program.

To link this toolkit into a QLiberator compiled program, use the following compiler directive somewhere near the start of the program to be compiled. The drive name is where the compiler can find the toolkit file.

```
110 REMark $$asmb=FLP2_DJToolkit_BIN,0,12
```

1. COPYRIGHT NOTICE AND DISCLAIMER

2. QUESTIONS ABOUT THE TOOLKIT

2.1 WHAT IS A TOOLKIT?

A toolkit is a set of BASIC extensions, new commands and functions which add to the number of "words" understood by the QL's BASIC language. These new extensions greatly add to the power and versatility of SuperBASIC, by adding the facility to perfrom new actions, or by simplifying a task which is difficult to program at the moment

2.2 CAN I USE THE TOOLKIT IN MY OWN PROGRAMS?

Yes, you can use these commands both in interpreted BASIC programs and compiled BASIC programs, but see the note above regarding use with Turbo.

2.3 DOES IT WORK WITH OTHER TOOLKITS?

We have tried to ensure that there is no clash, but it is impossible to test it against every other piece of software or hardware. Please let us know if you discover any incompatibilities so that we can try to sort them out.

3. BRIEF DESCRIPTION OF THE NEW COMMANDS

ABS_POSITION	Set file position absolute.
BYTES_FREE	How much free memory is left, in bytes.
CHECK	Test to see if a machine code PROC/FN exists.
DEV_NAME	Scan the Directory Device list, returning the next name.
DISPLAY_WIDTH	How many bytes are used to hold one screen line ?
DJ_OPEN	Opens a file, returns error or channel id.
DJ_OPEN_IN	Ditto, similar to OPEN_IN.
DJ_OPEN_NEW	Creates a file, returns channel id or error.
DJ_OPEN_OVER	Overwrites a file, returns error or channel id
DJ_OPEN_DIR	Opens a device directory for access.
DJTK_VER$	Return the toolkit version number as a string.
FETCH_BYTES	Get some bytes from a channel.
FILE_BACKUP	Get the backup date for a specific file.
FILE_DATASPACE	Get the file's dataspace.
FILE_LENGTH	Get the file's length.
FILE_POSITION	Get the current position in the file.
FILE_TYPE	Get the file's type.
FILE_UPDATE	Get the file's update date.
FILLMEM_B	Fill memory with a byte value.
FILLMEM_L	Fill memory with a long value.
FILLMEM_W	Fill memory with a word value.
FLUSH_CHANNEL	Flush the data on a channel to a device.
GET_BYTE	Fetch one byte from a channel.
GET_FLOAT	Fetch 6 bytes from a channel.
GET_LONG	Fetch 4 bytes from a channel.
GET_STRING	Fetch a QDOS string from a channel.
GET_WORD	Fetch 2 bytes from a channel.
KBYTES_FREE	How much free memory is left in Kbytes.
LEVEL2	Test whether level 2 drivers are present on a channel.
MAX_CON	Returns the absolute limits of a SCR or CON channel.
MAX_DEVS	Counts the number of directory devices. See DEV_NAME.
MOVE_MEM	Move memory around.
MOVE_POSITION	Set a file position relative to its current one.
PEEK_FLOAT	Read 6 bytes from memory into a float variable.
PEEK_STRING	Get bytes from memory into a string.
POKE_FLOAT	Pokes a floating point variable into memory.
POKE_STRING	Store the string in memory at a given address.
PUT_BYTE	Send 1 byte to a channel.
PUT_FLOAT	Send 6 bytes to a channel.
PUT_LONG	Send 4 bytes to a channel.
PUT_STRING	Send a QDOS string to a channel.
PUT_WORD	Send 2 bytes to a channel.
QPTR	Is the Pointer Environment available ?
READ_HEADER	Read the header for a file into a buffer.
RELEASE_HEAP	Remove some space allocated with RESERVE_HEAP.
RESERVE_HEAP	Get some Common Heap space for a program to use.
SCREEN_BASE	Find out where the screen memory starts for a channel.
SCREEN_MODE	Returns the current screen mode, 4 or 8.
SEARCH_C	Look in memory for a string, case is considered.
SEARCH_I	Ditto, but case is ignored.
SET_HEADER	Set the header for a file.
SET_XINC	Change horozontal spacing between characters.
SET_YINC	Change vertical spacing between lines of characters.
SYSTEM_VARIABLES	Find out where the system variables are.
USE_FONT	Change the fonts used by a channel.
WHERE_FONTS	Find the addresses of the two fonts used on a channel.

In the following descriptions, all parameters must be supplied as there are no defaults, in addition, when a channel number is being passed, either as a number or as a variable, it must be preceded by a hash (#).

4. THE NEW PROCEDURES

The following section gives details of all the new procedures, (not functions) in more detail. In the event of an error occurring in the procedures your program will stop with an error message.

ABS_POSITION #channel, position

This procedure will set the file pointer to the position given for the file attached to the given channel number. If you attempt to set the position for a screen or some other non-directory device channel, you will get a bad parameter error, as you will if position is negative.

If the position given is 0, the file will be positioned to the start, if the position is a large number which is greater than the current file size, the position will be set to the end of fileand no error will occur.

After an ABS_POSITION command, all file accesses will take place at the new position.

FILLMEM_B start_address, how_many, value
FILLMEM_W start_address, how_many, value
FILLMEM_L start_address, how_many, value

These procedures are provided so that a quick and simple way to fill up areas of memory is available to DJToolkit users. The 3 variations stuff bytes, words or long words into memory as required.

Note that the second parameter is the count of how many bytes, words or long words you want to put into the memory area starting at 'start_address'.

For example, the screen memory is 32 kilobytes long. To fill it all black, try this :-

```
1000 FILLMEM_B SCREEN_BASE(#0), 32 * 1024, 0
```

or this :-

```
1010 FILLMEM_W SCREEN_BASE(#0), 16 * 1024, 0
```

or this :-

```
1020 FILLMEM_L SCREEN_BASE(#0), 8 * 1024, 0
```

and the screen will change to all black. Note how the second parameter is halved each time, this is because there are half as many words as bytes and half as many longs as words.

The fastest is FILLMEM_L and the slowest is FILLMEM_B. When you use FILLMEM_W or FILLMEM_L you must make sure that the start_address is even or you will get a bad parameter error. FILLMEM_B does not care about its start_address being even or not.

FILLMEM_B truncates the value to the lowest 8 bits, FILLMEM_W to the lowest 16 bits and FILLMEM_L uses the lowest 32 bits of the value. Note that some values may be treated as negatives when PEEK'd back from memory. This is due to the QL treating words and long words as signed numbers.

FLUSH_CHANNEL #channel

This procedure makes sure that all data written to the given channel number has been 'flushed' out to the appropriate device. This means that if a power cut occurs, then no data will be lost.

MOVE_MEM source, destination, length

This procedure will copy the appropriate number of bytes from the given source address to the destination address. If there is an overlap in the addresses, then the procedure will notice and take the appropriate action to avoid corrupting the data being moved. Most moves will take place from source to destination, but in the event of an overlap, the move will be from (source + length -1) to (destination + length -1).

This procedure tries to do the moving as fast as possible and checks the addresses passed as parameters to see how it will do this as follows :-

1. If both addresses are odd, move one byte, increase the source & destination addresses by 1 and drop in to treat them as if both are even, which they now are !

2. If both addresses are even, calculate the number of long word moves (4 bytes at a time) that are to be done and do them. Now calculate how

many single bytes need to be moved (zero to 3 only) and do them.

3. If one address is odd and the other is even the move can only be done one byte at a time, this is quite a lot slower than if long words can be moved.

The calculations to determine which form of move to be done adds a certain overhead to the function and this can be the slowest part of a memory move that is quite small.

MOVE_POSITION #channel, relative_position

This is a similar procedure to ABS_POSITION, but the file pointer is set to a position relative to the current one. The direction given can be positive to move forward in the file, or negative to move backwards. The channel must of course be opened to a file on a directory device. If the position given would take you back to before the start of the file, the position is left at the start, position 0. If the move would take you past the end of file, the file is left at end of file.

After a MOVE_POSITION command, the next access to the given channel, whether read or write, will take place from the new position.

POKE_FLOAT address, value

This procedure will poke the 6 bytes that the QL uses to represent a floating point variable into memory at the given address. The address can be odd or even as the procedure can cope either way.

POKE_STRING address, string

This procedure simply stores the strings contents at the given address. Only the contents of the string are stored, the 2 bytes defining the length are not stored. The address may be odd or even.

If the second parameter given is a numeric one or simply a number, beware, QDOS will convert it to the format that would be seen if the number was PRINTed before storing it at the address. For example, 1 million would be '1E6' which is arithmetically the same, but characterwise, very different.

PUT_BYTE #channel, byte

The given byte is sent to the channel. If a byte value larger than 255 is given, only the lowest 8 bits of the value are sent. The byte value written to the channel will always be between 0 and 255 even if a negative value is supplied. GET_BYTE returns all negative values as positive.

PUT_FLOAT #channel, float

The given float value is converted to the internal QDOS format for floating point numbers and those 6 bytes are sent to the given channel number. The full range of QL numbers can be sent including all the negative values. GET_FLOAT will return negative values correctly (unless an error occurs).

PUT_LONG #channel, long

The long value given is sent as a sequence of four bytes to the channel. Negative values can be put and these will be returned correctly by GET_LONG unless any errors occur.

PUT_STRING #channel, string

The string parameter is sent to the appropriate channel as a two byte word giving the length of the data then the characters of the data. If you send a string of zero length, LET A$ = "" for example, then only two bytes will be written to the file. See POKE_STRING for a description of what will happen if you supply a number or a numeric variable as the second parameter. As with all QL strings, the maximum length of a string is 32kbytes.

PUT_WORD #channel, word

The supplied word is written to the appropriate channel as a sequence of two bytes. If the word value supplied is bigger than 65,535 then only the lower 16 bits of the value will be used. Negative values will be returned by GET_WORD as positive.

RELEASE_HEAP address

The address given is assumed to be the address of a chunk of common heap as allocated earlier in the program by RESERVE_HEAP. In order to avoid crashing the QL when an invalid address is given, RELEASE_HEAP check first that there is a flag at address-4 and if so, clears the flag and returns the memory back to the system. If the flag is not there, or if the area has already been released, then a bad parameter error will occur.

It is more efficient to RELEASE_HEAP in the opposite order to that in which it was reserved and will help to avoid heap fragmentation.

SET_XINC #channel, increment
SET_YINC #channel, increment

These two functions change the spacing between characters horozontally, SET_XINC, or vertically, SET_YINC. This allows slightly more information to be displayed on the screen. SET_XINC allows adjacent characters on a line of the screen to be positioned closer or further apart as desired. SET_YINC varies the spacing between the current line of characters and the next.

By choosing silly values, you can have a real messy screen, but try experimenting with OVER as well to see what happens. Use of the MODE or CSIZE commands in SuperBasic will overwrite your new values.

USE_FONT #channel, font1_address, font2_address

This is a procedure that will allow your programs to use a character set that is different from the standard QL fonts. The following example will suffice as a full description :-

```
1000 REMark Change the character set for channel #1
1010 :
1020 REMark Reserve space for the font file
1030 size = FILE_LENGTH('flp1_font_file')
1040 IF size < 0
1050    PRINT 'Font file error ' & size
1060    STOP
1070 END IF
1080 :
1090 REMark Reserve space to load font into
1200 font_address = RESERVE_HEAP(size)
1210 IF font_address < 0
1220    PRINT 'Heap error ' & font_address
1230    STOP
1240 END IF
1250 :
1260 REMark Load the font
1270 LBYTES flp1_font_file, font_address
1280 :
1290 REMark Now use the new font
1300 USE_FONT #1, font_address, 0
```

.....Rest of program

```
9000 REMark Reset channel #1 fonts
9010 USE_FONT #1, 0, 0
9020 :
9030 REMark Release the storage space
9040 RELEASE_HEAP font_address
```

In the above example, only one font has been changed for the channel, there are usualy 2 fonts in use on each channel. The procedure allows you to change 1 or both depending upon the application and to reset them back to the standard fonts at any time. The value of zero for any of the addresses will switch to the standard font found in the QL's ROM.

Many QL programs come with a number of font files, however, do not try to use the special type of fonts that come with many drawing or desk top publishing programs, such as Page Designer. These can sometimes be recognised by the file name which will include something like '_hires' or '_hdf' in the name. Ordinary QDOS font files as supplied with Digital Precision's Lightning can be loaded and used, as can the normal font files from Page Designer and the like.

5. THE NEW FUNCTIONS

The following section gives details of the new functions. Note that all functions which return a numeric value may also return a QDOS error code. This is usually a negative number. However, you should be aware that the functions GET_LONG and GET_FLOAT can return negative values as a valid result. There is no way to determine whether the negative result is a valid long or float, or if an error occurred.

Functions that return strings, for example DEV_NAME, cannot return an error code, so if any errors occur in them, they will act in a similar way to procedures and cause the program to stop with an appropriate error message.

Any function can cause an error that will stop your program from running if something goes wrong when the function tries to fetch its parameters, in this case the function will simply 'fall over' and QDOS will stop your program with an error. This should actually never happen, but then again ...

I have tried to make the functions as friendly as possible so that an error code is returned whenever possible, but if something does go wrong while fetching the parameters, who knows what state the maths stack is in and so it is best to let QDOS handle the problem.

oops = CHECK('name')

If name is a currently loaded machine code procedure or function, then the variable oops will be set to 1 otherwise it will be set to 0. This is a handy way to check that an extension command has been loaded before calling it. In a Turbo'd or Supercharged program, the EXEC will fail and a list of missing extensions will be displayed, a QLiberated program will only fail if the extension is actually called.

memory = BYTES_FREE

This simple function returns the amount of memory known by the system to be free. The answer is returned in bytes, see also KBYTES_FREE below. For the technically minded, the free memory is considered to be that between the addresses held in the system variables SV_FREE and SV_BASIC.

device$ = DEV_NAME(address)

This function must be called with a floating point variable name as its parameter. The first time this function is called, address must hold the value zero, on all other calls, simply pass address UNCHANGED back. The purpose of the function is to return a directory device name to the variable device$, an example is worth a thousand explanations.

```
1000 addr = 0
1010 REPeat loop
1020   PRINT "<" & DEV_NAME(addr) & ">"
1030   IF addr = 0 THEN EXIT loop: END IF
1040 END REPeat loop
```

This small example will scan the entire directory device driver list and return one entry from it each time as well as updating the value in 'addr'. The value in addr is the start of the next device driver linkage block and MUST NOT BE CHANGED except by the function DEV_NAME. If you change addr and then call DEV_NAME again, the results will be very unpredictable.

The check for addr being zero is done as this is the value returned when the final device name has been extracted, in this case the function returns an empty string for the device. If the test was made before the call to DEV_NAME, nothing would be printed as addr is zero on entry to the loop.

Please note, every QL has at least one device in the list, the 'MDV' device and some also have a device with no name as you will see if you run the above example (not the last one as it is always an empty string when addr becomes zero).

The above example will only show directory devices, those that can have DIR used on them, or FORMAT etc, such as WIN, RAM, FLP, FDK etc, however, it cannot show the non-directory devices such as SER, PAR or NUL, if you have Lightning, as these are in another list held in the QL.

From version 1.14 of DJToolkit onwards, there is a function that counts the number of directory devices present in the QL. See MAX_DEVS below for details.

bytes_in_a_line = DISPLAY_WIDTH

This function can be used to determine how many bytes of the QL's memory are used to hold the data in one line of pixels on the screen. Note that the value returned has nothing to do with any window width, it always refers to the total screen display width.

Why include this function I hear you think ? If you run an ordinary QL, then the result will probably always be 128 as this is how many bytes are used to hold a line of pixels, however, many people use Atari ST/QLs and these have a number of other screen modes for which 128 bytes is not enough. This function will return the exact number of bytes required to step from one line of pixels to the next. Never assume that QDOS programs will only

ever be run on a QL. What will happen when new Graphics hardware arrives ? This function will still work, assuming that the unit uses standard QDOS channel definition blocks etc.

For the technically minded, the word at offset $64 in the SCR_ or CON_ channel's definition block is returned. This is called SD_LINEL in 'Tebby Speak' and is mentioned in Jochen Merz's QDOS Reference Manual and the QL Technical Manual by Tony Tebby et al. Andrew Pennel's book, the QDOS Companion gets it wrong on page 61, guess which one I used first !

v$ = DJTK_VER$

This simply sets v$ to be the 4 character string 'n.nn' where this gives the version number of the current toolkit. If you have problems, always quote this number when requesting help.

channel = DJ_OPEN('filename') [open existing file for exclusive use]
channel = DJ_OPEN_IN('filename') [open existing file for shared use]
channel = DJ_OPEN_NEW('filename') [create a new file foe exclusive use]
channel = DJ_OPEN_OVER('filename') [overwrite an existing file]
channel = DJ_OPEN_DIR('filename') [open device directory or sub-directory]

All of these functions return the SuperBasic channel number if the channel was opened ok. If any error occurred, the channel number will be negative. You can use this to check whether the open was successful or not. The filename must be supplied as a variable name, file$ for example, or in quotes, 'flp1_fred_dat'.

They all work in a similar manner to the normmal SuperBasic OPEN procedures, but, DJ_OPEN_DIR is new.

I am grateful to Simon N. Goodwin for his timely article in QL WORLD volume 2, issue 8 (marked Vol 2, issue 7 !!!). I had been toying with these routines for a while and was aware of the undocumented QDOS routines to extend the SuperBasic channel table. I was, however, not able to get my routines to work properly. Simon's article was a great help and these functions are based on that article. Thanks Simon.

The OPEN routines work as follows :

```
1000 REMark open our file for input
1010 :
1020 chan = DJ_OPEN_IN('filename')
1030 IF chan < 0
1040    PRINT 'OOPS, failed to open "filename", error ' & chan
1050    STOP
1060 END IF
1070 :
1080 REM process data in file here ....
```

DJ_OPEN_DIR is a new function to those in the normal QL range, and it works as follows :-

```
1000 REMark read a directory
1010 :
1020 INPUT 'Which device ';dev$
1030 chan = DJ_OPEN_DIR(dev$)
1040 IF chan < 0
1050    PRINT 'Cannot open ' & dev$ & ', error ' & chan
1060    STOP
1070 END IF
1080 :
1090 CLS
1100 REPeat dir_loop
1110   IF EOF(#chan) THEN EXIT dir_loop
1120   a$ = FETCH_BYTES(#chan, 64)
1130   size = CODE(a$(16)):       REMark Size of file name
1140   PRINT a$(17 TO 16 + size): REMark file name
1150 END REPeat dir_loop
1160 :
1170 CLOSE #chan
1180 STOP
```

In this example, no checks are done to ensure that the device actually exists, etc. You could use DEV_NAME to check if it is a legal device. The data being read from a device directory file must always be read in 64 byte chuncks as per this example.

Each chunk is a single directory entry which holds a copy of the file header for the appropriate file. Note, that the first 4 bytes of a file header hold the actual length of the file but when read from the directory as above, the value if 64 bytes too high as it includes the length of the file header as part

of the length of a file.

The above routine will also print blank lines if a file has been deleted from the directory at some point. Deleted files have a name length of zero.

Note that if you type in a filename instead of a device name, the function will cope. For example, you type in 'flp1_fred' instead of 'flp1_'. You will get a list of the files on 'flp1_' if 'fred' is a file, or even, if 'fred' is not on 'flp1_'. If, however, you have the LEVEL 2 drivers (see LEVEL2 below), and 'fred' is a sub-directory then you will get a listing of the sub-directory as requested.

a$ = FETCH_BYTES(#channel, how_many)

This function returns the requested number of bytes from the given channel which must have been opened for INPUT or INPUT/OUTPUT. It will work on CON_ channels as well, but no cursor is shown and the characters typed in are not shown on the screen. If there is an ENTER character, or a CHR$(10), it will not signal the end of input. The function will not return until the appropriate number of bytes have been read.

WARNING - JM and AH ROMS will cause a 'Buffer overflow' error if more than 128 bytes are fetched, this is a fault with QDOS and not with DJToolkit. See the demos file for a 'fix' to this problem.

bk = FILE_BACKUP(#channel)
bk = FILE_BACKUP('filename')

This function reads the backup date from the file header and returns it into the variable bk. The parameter can either be a channel number for an open channel, or it can be the filename (in quotes) of a closed file. If the returned value is negative, it is a normal QDOS error code. If the value returned is positve, it can be converted to a string be calling DATE$(bk). In normal use, a files backup date is never set by QDOS, however, users who have WinBack or a similar backup utility program will see proper backup dates if the file has been backed up.

ds = FILE_DATASPACE(#channel)
ds = FILE_DATASPACE('filename')

This function returns the current dataspace requirements for the file opened as #channel or for the file which has the name given, in quotes, as filename. If the file is an EXEC'able file (See FILE_TYPE) then the value returned will be the amount of dataspace that that program requires to run, if the file is not an EXEC'able file, the result is undefined, meaningless and probably zero. If the result is negative, there has been an error and the QDOS error code has been returned.

fl = FILE_LENGTH(#channel)
fl = FILE_LENGTH('filename')

The actual file length is returned, as above, the file may be open, in which case simply supply the channel number, or closed, supply the filename in quotes. If the returned value is negative, then it is a QDOS error code.

where = FILE_POSITION(#channel)

This function will tell you exactly where you are in the file that has been opened, to a directory device, as #channel, if the result returned is negative it is a QDOS error code. If the file has just been opened, the result will be zero, if the file is at the very end, the result will be the same as calling FILE_LENGTH(#channel) - 1, files start at byte zero remember.

ft = FILE_TYPE(#channel)
ft = FILE_TYPE('filename')

This function returns the files type byte. The various types currently known to me are :

0 = BASIC, CALL'able machine code, an extensions file or a DATA file.
1 = EXEC'able file.
2 = SROFF file used by linkers etc, a C68 Library file etc.
3 = THOR hard disc directory file. (I think !)
4 = A font file in The Painter
5 = A pattern file in The Painter
6 = A compressed MODE 4 screen in The Painter
11 = A compressed MODE 8 screen in The Painter
255 = Level 2 driver directory or sub-directory file, Miracle hard disc directory file.

There may be others.

fu = FILE_UPDATE(#channel)
fu = FILE_UPDATE('filename')

This function returns the date that the appropriate file was last updated, either by PRINTING to it, SAVING it or editing it using an editor etc. This date is set in all known QLs.

byte = GET_BYTE(#channel)

Reads one character from the file attached to the channel number given and returns it as a value between 0 and 255. This is equivalent to CODE(INKEY$(#channel)). BEWARE, PUT_BYTE can put negative values to file, for example -1 is put as 255, GET_BYTE will return 255 instead of -1. Any negative numbers returned are always error codes.

float = GET_FLOAT(#channel)

Reads 6 bytes from the file and returns them as a floating point value. BEWARE, if any errors occur, the value returned will be a negative QDOS error code. As GET_FLOAT does return negative values, it is difficult to determine whether that returned value is an error code or not. If the returned value is -10, it could actually mean End Of File, this is about the only error code that can be tested for.

long = GET_LONG(#channel)

Read the next 4 bytes from the file and return them as a number between 0 and 2^32 -1 (4,294,967,295 or HEX FFFFFFFF unsigned). The returned value is equivalent to the following fragment of SuperBasic code :

```
2100 REMark Get a long word from the open channel
2110 long = 0
2120 FOR x = 1 TO 4
2130   IF EOF(#channel) THEN EXIT x: END IF
2140   long = long * 256 + CODE(INKEY$(#channel))
2150 END FOR x
```

BEWARE, the same problem with negatives & error codes applies here as well as GET_FLOAT.

a$ = GET_STRING(#channel)

Read the next 2 bytes from the file and assuming them to be a QDOS string's length, read that many characters into a$. The two bytes holding the string's length are NOT returned in a$, only the data bytes. The subtle difference between this function and FETCH_BYTES is that this one finds out how many bytes to return from the channel given, FETCH_BYTES needs to be told how many to return by the user. GET_STRING is the same as FETCH_BYTES(#channel, GET_WORD(#channel)).

WARNING - JM and AH ROMS will give a 'Buffer overflow' error if the length of the returned string is more than 128 bytes. This is a fault in QDOS, not DJToolkit. The demos file has a 'fix' for this problem.

word = GET_WORD(#channel)

The next two bytes are read from the appropriate file and returned as an integer value. This is equivalent to CODE(INKEY$(#channel)) * 256 + CODE(INKEY$(#channel)). See the caution above for GET_BYTE as it applies here as well. Any negative numbers returned will always be an error code.

memory = KBYTES_FREE

The amount of memory considered by QDOS to be free is returned rounded down to the nearest kilo byte. See BYTES_FREE above also. The value in KBYTES_FREE may not be equal to BYTES_FREE/1024 as the value returned by KBYTES_FREE has been rounded down.

present = LEVEL2(#channel)

If the device that has the given channel opened to it has the level 2 drivers, then present will be set to 1, otherwise it will be set to 0. The level 2 drivers allow such things as sub_directories to be used, when a DIR is done on one of these devices, sub-directories show up as a filename with '->' at the end of the name. Gold Cards and later models of Trump cards have level 2 drivers. Microdrives don't.

error = MAX_CON(#channel%, x%, y%, xo%, yo%)

If the given channel is a 'CON_' channel, this function will return a zero in the variable 'error'. The integer variables, 'x%', 'y%', 'xo%' and 'yo%' will be altered by the function, to return the maximum size that the channel can be WINDOW'd to.

'x%' will be set to the maximum width, 'y%' to the maximum depth, 'xo%' and 'yo%' to the minimum x co-ordinate and y co-ordinate respectively.

For the technically minded reader, this function uses the IOP_FLIM routine in the pointer Environment code, if present. If it is not present, you should get the -15 error code returned. (BAD PARAMETER).

how_many = MAX_DEVS

This function returns the number of installed directory device drivers in your QL. It can be used to DIMension a string array to hold the device names as follows :-

```
1000 REMark Count directory devices
1010 :
1020 how_many = MAX_DEVS
1030 :
1040 REMark Set up array
1050 :
1060 DIM device$(how_many, 10)
1070 :
1080 REMark Now get device names
1090 addr = 0
1100 FOR devs = 1 to how_many
1110   device$(devs) = DEV_NAME(addr)
1120   IF addr = 0 THEN EXIT devs: END IF
1130 END FOR devs
```

As there is a secondary check to ensure that we don't exceed the list of devices, line 1120, you can, if you wish, DIM the array slightly larger than required. The above routine will let you fill up the array DEVICE$ with the names of all the currently installed directory devices. (FLP, RAM etc). Note that on some (all ?) QLs there seems to be a device with a null name.

value = PEEK_FLOAT(address)

This function returns the floating point value represented b y the 6 bytes stored at the given address. BEWARE, although this function cannot detect any errors, if the 6 bytes stored at 'address' are not a proper floating point value, the QL can crash. The crash is caused by QDOS and not by PEEK_FLOAT. This function should be used to retrieve values put there by POKE_FLOAT mentioned above.

a$ = PEEK_STRING(address, length)

The characters in memory at the given address are returned to a$. The address may be odd or even as no word for the length is used, the length of the returned string is given by the length parameter. This is the opposite to the POKE_STRING command described above and equivalent to the following SuperBasic function :

```
2700 DEFine FuNction PEEK_STRING(address, length)
2720   LOCal x, result$
2730   result$ = ''
2740   FOR x = 0 TO length - 1
2750     result$ = result$ & CHR$(PEEK(address + x))
2760   END FOR x
2770   RETurn result$
2780 END DEFine PEEK_STRING
```

PE_Found = QPTR(#channel)

This function returns 1 if the Pointer Environment is loaded or 0 if not. The channel must be a SCR_ or CON_ channel, if not, the result will be 0. If a silly value is given then a QDOS error code will be returned instead.

error = READ_HEADER(#channel, buffer)

The file that is opened on the given channel has its header data read into memory starting at the given address (buffer). The buffer address must have been reserved using RESERVE_HEAP, see below, or some similar command. The buffer must be at least 64 bytes long or unpredictable

results will occur. The function will read the header but any memory beyond the end of the buffer will be overwritten if the buffer is too short. After a successful call to this function, the contents of the buffer will be as follows :

Buffer + 0	file length	4 bytes long (see FILE_LENGTH)
Buffer + 4	file access	1 byte long - currently zero
Buffer + 5	file type	1 byte long (see FILE_TYPE)
Buffer + 6	file dataspace	4 bytes long (see FILE_DATASPACE)
Buffer + 10	unused	4 bytes long
Buffer + 14	name length	2 bytes long, size of filename
Buffer + 16	filename	36 bytes long

Directory devices also have the following additional data :

Buffer + 52	update date	4 bytes long (see FILE_UPDATE)
Buffer + 56	reference date	4 bytes long - see below
Buffer + 60	backup date	4 bytes long (see FILE_BACKUP)

Miracle Systems hard disc's users and level 2 users will find the files version number stored as the the 2 bytes starting at buffer + 56, the remaining 2 bytes of the reference date seem to be hex 094A or decimal 2378 which has no apparent meaning, this of course may change at some point !

This function returns an error code if something went wrong while attempting to read the file header or zero if everything went ok. It can be used as a more efficient method of finding out the details for a particular file rather than calling all the various FILE_XXXXX functions. Each of these call the READ_HEADER routine.

To extract data, use PEEK for byte values, PEEK_W for the filename length and version number (if level 2 drivers are present, see LEVEL2 above), or PEEK_L to extract 4 byte data items.

The filename can be extracted from the buffer by something like
`f$ = PEEK_STRING(buffer + 16, PEEK_W(buffer + 14)).`

buffer = RESERVE_HEAP(length)

This function obtains a chunk of memory for your program to use, the starting address is returned as the result of the call. Note that the function will ask for 4 bytes more than you require, these are used to store a flag so that calls to RELEASE_HEAP, see above, do not crash the system by attempting to deallocate invalid areas of memory. If you call this function, the returned address is the first byte that your program can use. It can be used as follows to reserve a buffer for READ_HEADER, described above.

```
1000 buffer = RESERVE_HEAP(64)
1010 IF buffer < 0
1020    PRINT 'ERROR allocating buffer, ' & buffer
1030    STOP
1040 error = READ_HEADER(#3, buffer)
.....do something with buffer contents here
2040 REMark Finished with buffer
2050 RELEASE_HEAP buffer
```

screen = SCREEN_BASE(#channel)

This function is handy for Minerva users, who have 2 screens to play with. The function returns the address of the start of the screen memory for the appropriate channel.

If the returned address is negative, consider it to be a QDOS error code. (-6 means channel not open & -15 means not a SCR_ or CON_ channel.)

SCREEN_BASE allows you to write programs that need not make guesses about the whereabouts of the screen memory, or assume that if VER$ gives a certain result, that a Minerva ROM is being used, this may not always be the case. Regardless of the ROM in use, this function will always return the screen address for the given channel.

current_mode = SCREEN_MODE

This function can help in your programs where you need to be in a specific mode. If you call this function you can find out if a mode change needs to be made or not. As the MODE call changes the mode for every program running in the QL, use this function before setting the appropriate mode. The value returned can be 4 or 8 for normal QLs, 2 for Atari ST/QL Extended mode 4 or any other value deemed appropriate by the hardware being used. Never assume that your programs will only be run on a QL ! (Graphics Card anyone ?)

```
1000 REMark Requires MODE 4 for best results so ...
1010 IF SCREEN_MODE = 8
```

```
1020    MODE 4
1030 END IF
1040 :
1050 REMark Rest of program ....
```

address = SEARCH_C(start, length, what_for$)
address = SEARCH_I(start, length, what_for$)

Both of these functions search through memory looking for the given string. SEARCH_C searches for an EXACT match whereas SEARCH_I ignores the difference between lower & UPPER case letters.

If the address returned is zero, the string was not found, otherwise it is the address where the first character of what_for$ was found, or negative for any errors that may have occurred. An example :

```
1000 PRINT SEARCH_C(0, 48 * 1024, 'sinclair')
1010 PRINT SEARCH_I(0, 48 * 1024, 'sinclair')
1020 :
1030 PRINT SEARCH_C(0, 48 * 1024, 'Sinclair')
1040 PRINT SEARCH_I(0, 48 * 1024, 'Sinclair')
```

The above fragment, on my Gold Card JS QL, prints :

0 at line 1000 - case must match exactly, no capital 'S'
47314 at line 1010 - case ignored
47314 at line 1030 - 'Sinclair' found exactly
47314 at line 1040 - case ignore again

Looking into the ROM at that address using PEEK_STRING(47314, 21) gives :

Sinclair Research Ltd

which is part of the copyright notice that comes up when you switch on your QL.

If the string being searched for is empty ("") then zero will be returned, if the length of the buffer is negative or 0, you will get a 'bad parameter' error (-15). The address is considered to be unsigned, so negative addresses will be considered to be very large positive addresses, this allows for any future enhancements which will allow the QL to use a lot more memory than it does now !

error = SET_HEADER(#channel, buffer)

This function returns the error code that occurred when trying to set the header of the file on the given channel, to the contents of the 64 byte buffer stored at the given address. If the result is zero then you can assume that it worked ok, otherwise the result will be a negative QDOS error code. On normal QLs, the three dates at the end of a file header cannot be set.

sys_vars = SYSTEM_VARIABLES

This function returns the current address of the QL's system variables. For most purposes, this will be hex 28000, decimal 163840, but Minerva users will probably get a different value due to the double screen. DO NOT assume that all QLs, current or future, will have their system variables at a fixed point in memory, this need not be the case.

address = WHERE_FONTS(#channel, 1_or_2)

This function returns a value that corresponds to the address of the fonts in use on the specified channel. The second parameter must be 1 for the first font address or 2 for the second, there are two fonts used on each channel. If the result is negative then it will be a normal QDOS error code. The channel must be a CON_ or a SCR_ channel to avoid errors.

6. QDOS ERROR CODES

Many of the above functions return a valid result, such as an address, or a negative error code. The QDOS error codes are listed below for reference.

-1 Not complete
-2 Invalid job
-3 Out of memory
-4 Out of range
-5 Buffer overflow
-6 Channel not open
-7 Not found
-8 File already exists
-9 In use
-10 End of file
-11 Drive full
-12 Bad device name
-13 Xmit (transmit) error
-14 Format failed
-15 Bad parameter
-16 File error
-17 Error in expression
-18 Arithmetic overflow
-19 Not implemented
-20 Read only
-21 Bad line

7. DJTOOLKIT BASIC DEMONSTRATION ROUTINES 1

BY DILWYN JONES, 1993

This document gives some instructions for the demonstration routines supplied with the DJToolkit. They are contained in the file DEMOS_bas, which can be loaded from BASIC with the LOAD command.

FIRST A WARNING FROM NORMAN DUNBAR

You may be aware of a QL BUG which causes problems with large numbers of parameters and/or local variables. I have detailed it here as well, just in case.

Some of the demo procedures and functions have a large number of parameters and/or local variables. When the total of these comes to more than 9 the QL will probably crash when these routines are executed. To avoid this, I have REMarked out the 'extra' LOCals in some of the routines, however, if you compile them and ONLY run the compiled version, you can unREMark them as most compilers fix this bug. Nowback to Dilwyn's instructions for his demo routines.

Those short programs, implemented as procedures and functions, are intended to give some examples of how to use Toolkit extensions, or in some cases how to get around some limitations or how to adapt the useage slightly in practical use or on certain ROM versions where results may vary slightly. These are, in effect, "programmed routines". Most are simple demonstrations only, some can easily be adapted or copied for use in your programs, while one (FIND_FILE) is a really useful, complete program in its own right.

This document is supplied as a Quill _DOC file for two reasons. Firstly, since the price of the DJToolkit is quite low, it saves on costs by not having to print a larger manual than is necessary for the toolkit itself. Secondly, it allows us to change or add to these example routines as required at short notice, or to fix bugs, without the expense and inconvenience of having to reprint instructions all the time.

The instructions are listed in the order in which the routines are listed in DEMOS_bas, using the name of the procedure or function. Users who have the QREF utility from Liberation Software (available from DJC) can use the QFIND utility to quickly locate the routine.

These routines can be used in compiled commercial software along with the toolkit - follow the guidelines in the manual.

Beware - some routines call others in this file, e.g. the CURSOR_ENABLE and CURSOR_DISABLE routines are used by some of the routines.

PLEASE NOTE: The routines in DEMOS1_bas were written using an older version of DJToolkit. The routines which directly access the display, for example, could be improved through the use of MAX_CON, for example, to test the maximum sizes of display available.

7.1 CURSOR_ENABLE

This routine shows how to use CHECK to test if given commands are present in the machine and vary its action accordingly. Since DJToolkit has no cursor enabling command of its own (normally, you'd use the Toolkit 2 CURSEN or the QLiberator Q_CURSON) this routine finds out which version is present and switches on the cursor in the specified channel. If neither is available, it uses an undocumented command present in current ROMs, but which may not be implemented in future ROM versions or operating systems such as SMS. You could extend this routine to look for other cursor enabling commands, or adapt it to search for other types of extensions to allow your programs to take advantage of toolkits loaded or to report errors if toolkits required are not present, so that a program can stop neatly if an extension is not present rather than just give an obscure error message. One parameter is required, the channel number in which you want a cursor to be enabled. Note that just because a cursor has been enabled, it does not necessarily follow that it will be flashing! Indeed, this action may vary from ROM version to ROM version - you may have to press CTRL C (hold down CTRL and tap the C key) until the right cursor starts flashing, especially if there are several jobs running in the QL at the time. Why have a cursor flashing at all? Because (unless you are using the Extended Environment, as supplied with software such as QPAC2) you can only have keyboard input to a compiled program if the cursor is enabled before reading. INPUT does not need you to do this, since it turns on the cursor by itself, but INKEY$ does need a cursor to be enabled. If you have a compiled program which uses INKEY$ and you CTRL C out of it to another program, you can be locked out of the program.

7.2 CURSOR_DISABLE

Performs the opposite action to the CURSOR_ENABLE routine above. It stops the cursor flashing and turns it off. Here's a couple of short examples of how to use these two routines:-

```
REMark wait for a keypress, PAUSE works from #0 in older ROMs
PRINT #0,'PRESS A KEY TO CONTINUE. ';
CURSOR_ENABLE #0 : PAUSE : CURSOR_DISABLE #0 : PRINT #0,

REMark using INKEY$
CURSOR_ENABLE #3 : LET k$ = INKEY$(#3) : CURSOR_DISABLE #3
```

Note that if INKEY$ is used in a loop, rapidly switching the cursor off and on may cause the cursor to flicker annoyingly, so either use a delay in

INKEY$ or put it in a loop with the cursor controls outside the loop, e.g.

```
REMark wait for a key to be pressed
CURSOR_ENABLE #0
REPeat get_key
  key = CODE(INKEY$)
  IF key <> 0 : EXIT get_key
END REPeat get_key
CURSOR_DISABLE #0

REMark another method
CURSOR_ENABLE #0
key = CODE(INKEY$(-1)) : REMark INKEY$ defaults to #0
CURSOR_DISABLE #0
```

Note for when compiling programs - INPUT defaults to screen channel #1 (normally the red part of the screen after starting up the QL), while INKEY$ and PAUSE default to channel #0 (the part at the bottom of the screen). INPUT and INKEY$ can both have specified channel numbers (INKEY$(#number) and INPUT#number), but PAUSE cannot in most versions of the ROM. PAUSE can take an optional channel number under a Minerva ROM. If you need a delay and are using a window other than channel #0, there may be difficulties with PAUSE on an older ROM, so use INKEY$ instead. PAUSE n is roughly equivalent to LET variable$ = INKEY$(#0,n), so it is easy to make INKEY$ work from another channel. The string variable is only there to accept the dummy result from INKEY$ since that is a function - sometimes it is useful for detecting the ESC key, which returns a CHR$(27):

```
PRINT#0,'PRESS A KEY TO CONTINUE, OR ESC TO QUIT ';
CURSOR_ENABLE #0
LET k$ = INKEY$(#0,-1) : REMark wait until key pressed
CURSOR_DISABLE #0
IF k$ = CHR$(27) : STOP : REMark ESC key pressed
```

You will notice I took a lot of space to explain cursors! When compiling programs, it is one of the biggest sources of errors and difficulties new programmers encounter when attempting to ensure that their programs can multi-task or task switch properly. Since DJToolkit is intended mainly for use with QLiberator, I thought it was the least I could do to help with these problems!

7.3 SLIDE_SHOW

This routine loads a number of QL screen pictures from disk and stores them in memory, then shows them on screen one after another like a slide show. Each QL screen picture has to be in the same mode (i.e. mode 4 or mode 8 or other 32k mode supported by your hardware). Since each picture requires 32k of memory, it is a routine only likely to be useful on a machine with expanded memory. First, the routine asks for the number of pictures to be loaded, then tries to reserve enough memory in the common heap (if there is not enough free memory, the program stops). Next, it asks for the filenames of the screens to be loaded. Enter those (remember to press ENTER after each one of course). After each filename is entered, the screen is loaded into a slot in the common heap. Note the rather strange looking expression in the LBYTES statement - the '&""' expression is needed since sc$ is a string array and in versions AH, JM and JS, using a string array as a parameter for LBYTES without having Toolkit 2 enabled can cause problems (unless of course you copied the string array to a normal undimensioned string variable first). Adding a null string to the array entry converts it to an expression rather than a standard array as far as LBYTES is concerned and gets around the problem. The fix is not needed if Toolkit 2 is used, since the redefined commands in Toolkit 2 do not exhibit this problem.

The code in the loop then copies the pictures to the screen one at a time with a slight pause between each one. The MOVE_MEM statement copies the 32k (32768 bytes) of each picture direct to the screen, while the SCREEN_BASE statement ensures it goes to the correct screen for that channel (e.g. if using Minerva second screen).

Finally, the RELEASE_HEAP statement frees up the memory used to hold the pictures.

7.4 REFLECT_SCREEN

This routine simply reflects the top half of the screen into the bottom half, like a mirror effect. The variable "from" starts from the top of the current screen (note the use of SCREEN_BASE to check where the screen is), while the variable "dest" points to the start of the last line of the screen, to work up from the bottom. The routine works by copying the top line of the screen to the bottom, then the second line is moved to the bottom line but one, and so on. PEEK_STRING and POKE_STRING are used to copy chunks of memory - it is possible to use MOVE_MEM to do this as well. Sadly, it is not possible to create a routine to reflect sideways in the same way due to the way the screen is laid out. Note how this routine uses the DISPLAY_WIDTH function to ensure that whole lines are moved at a time (normally on a standard QL, the standard width of a line in bytes is 128, but it can be different on new hardware, so using techniques such as this should help to ensure such programs work on new display hardware). You may like to adapt the maximum display sizes routine below to also allow the maximum height of the screen to be calculated. Note that this routine uses another function in this demo file, DISPLAY_WIDTH_JM (see below for details) to ensure that the routine can work properly on version AH and JM QLs.

7.5 ZOOM_IN

This routine magnifies the top half of the screen picture, doubling its height to fill the whole screen. It works in a very similar way to the previous example, including the use of DISPLAY_WIDTH.

7.6 SAVE_THE_SCREEN

This routine saves the content of the screen so that it can be restored later. The routine reserves 32 kilobytes in the common heap then later restores it later as required, by using MOVE_MEM. You can adapt this routine for use in your own programs.

7.7 RANDOM_ACCESS

This demonstrates the use of the file handling functions to make files with fixed-length information in them, which can be read back later in a simple fashion, because the known length of the items makes it easy to fetch the information at known or predictable positions.

As an example, consider writing random numbers to a file. The number '100' is three digits long, while the number '6' is only one digit long. If the file is full of numbers of random length, we can't easily work out where each one starts. However, by using the PUT_BYTE, PUT_WORD or PUT_LONG commands we can send numbers to a file as one byte, two byte or four byte long items respectively, making it easier to recover the numbers later when required, since each number will be the same length in the file.

The first loop generates 100 random numbers in the range 0 to 255 in value and sends them to a file with the PUT_BYTE command. Each byte can only hold numbers from 0 to 255 in value.

The second loop reads back the random numbers from the file. You are asked to enter a number from 1 to 100 to choose any of the numbers from the file (if you enter 0 the routine stops). Each entry is one byte long, so to fetch the fortieth number, for example, we position the file pointer to the start of the fortieth entry, remembering that in common with many computer conventions, files start at position 0, so the fortieth item is actually at position 39. The ABS_POSITION command is used to move the file pointer to the required position. Once the pointer is at the required location, we use GET_BYTE to fetch the number and it is then displayed on the screen.

Although this is a trivial example of random access files (the name has nothing to do with the random numbers used, it merely means that you can fetch any item you want from any point in the file), it does serve to illustrate some of the techniques used in database programs. In these, the technique often used is to specify in advance the maximum size of each record (or entry). Compare this with the way in which you define screens in Archive. Provided a little care is used to ensure that each record written to the disk is the same length, it becomes a simple matter to fetch any given one by using multiples of this length to position the file pointer. If you add something to the database, it is a simple matter of adding it at the end or in an unused slot in the file. To change or alter a record, fetch it from the disk, edit it then put it back where it came from. To delete a record, simply mark it as unused, then it can be re-used when something new is added.

In case you have not come across it before, a record is one whole entry in a file. Imagine you have a record collection - that is the file. Each part is (surprise, surprise) one record. These records are often further divided into tracks on a musical record, or fields in a computer record. The fields (divisions within a record) are not always the same size, but it is quite common for each record to have the same structure. Examples of records and fields in use might be an address database. One field might hold initials or forenames, a second might hold surnames, while the third might hold the address. This could be expected to be much longer than the others.

The whole subject of random access file handling is very complex and really needs a whole book by itself to explain it, so don't be surprised if you don't quite get the hand of it immediately!

7.8 FILE_DETAILS

As its name implies, this routine shows the details such as file length, file type, dates etc for a file whose name is passed to the procedure (e.g. FILE_DETAILS 'flp1_boot'). The dataspace is also shown for an executable file.

7.9 DRIVE_DETAILS

This routine shows how to use DEV_NAME to produce a listing of QL devices to choose from, then proceed to list the details of files on that drive. It also shows how to make decisions about the best way for a program to perform an action depending on whether or not some particular item is present or not, in this case if a ramdisk is present or not. The program lists the devices on the screen for you to select one by entering its number, then you are asked for a drive number, then a list of files is shown along with their details.

7.10 SHOW_QUARTER_SCREENS

A short utility to aid with viewing QL screen pictures. A quarter of each picture found on a disk is displayed in each corner of the screen allowing a quick viewing facility. It again shows how to use POKE_STRING, SCREEN_BASE, etc to more correctly access the screen. The second version (**QUARTER_SCREENS2**) shows a different way of approaching the display, namely by placing the top left quarter of each picture in four quarters of the screen. This routine (QUARTER_SCREENS2) is rather slow in interpreted BASIC.

7.11 MAKE_DIRECTORY

A short utility for those writing programs which expect to be able to make use of hard directories (as found on Gold Card systems, or on hard disk systems). From BASIC, one way of checking to see if the system on which the program is running is able to support hard directories is to check if the MAKE_DIR command is present. You could use CHECK('MAKE_DIR') to do this. But the best way is to use the LEVEL2 function to test. This shows a simple way to test if it is possible to create hard directories.

7.12 SET_MODE

The purists will tell you that if the computer is already in the screen mode you want it to be, you should not issue another mode command to change it to the same mode! So if your program needs to run in MODE 4, it is bad practice to issue a MODE 4 statement if the computer is already in mode 4. This routine checks if the computer is in another screen mode before issuing a MODE command. Simply put the mode number required after SET_MODE (e.g. SET_MODE 4).

7.13 AUTO_REPEAT

This little program shows the current setting of the auto repeat system variables. The delay is the time before the key pressed starts to repeat, i.e. if you press and hold down A, how long it takes before the A appears. The period is the time between repeats, i.e. in the example given, the time between the second and third A and so on. Standard values on a UK version JS system are 30 and 2 respectively. The program shows the settings currently used and allows you to change them, which may be useful if another program changes the settings and makes the keyboard too fast or too slow to use. This routine shows how to use the SYSTEM_VARIABLES function to access the system variables wherever they may be hiding in a machine. Most QDOS documentation lists system variable locations as offsets from the base of the block (i.e. how many bytes further than where they start). This is how they are accessed by this function, for example, if a particular system variable is 140 bytes from the base of the system variables area, it is simply added to the number returned by SYSTEM_VARIABLES.

7.14 SHOW_CAPS_LOCK

This is a routine which sits across the system variables constantly monitoring the state of the CAPS LOCK flag, and prints a message on the screen to say if the caps lock is on or off. It looks at the system variable at offset 136 and prints either CAPS ON if set, or CAPS OFF if reset. A modification to this routine would enable it to remember the last setting it saw and only print if it spotted a change, so that it is not printing all the time as in this example.

You can extend this routine to force caps lock off or on if you wish. What you need to do is to study the values normally stored in this variable and modify the content to be whatever you want. On my machine, the values given by PEEK_W(SYSTEM_VARIABLES+136) are:

Caps lock off : 0
Caps lock on : -256

To switch on caps lock from a program, you'd use POKE SYSTEM_VARIABLES+136,-256. To switch off caps lock, you'd use 0 instead of -256.

7.15 LAST_KEY

This routine looks at the sv.arbuf system variable for a number corresponding to the last key pressed. In some cases, the number may not be what you expected, especially if you press ALT with another key, but it can be of use sometimes if one compiled program wishes to monitor for a given key press while another program is in use, e.g. a screen dump routine could be programmed to watch out for a key press to activate a screen dump to a printer of a picture within another program. Imagine that the screen dump routine is called SCREEN_DUMP and that the key to activate it is the TAB key. We might write a small piece of code like this to wait for the TAB key to be pressed, then activate the screen dump routine, or stop if the ESC key was pressed.

```
REPeat get_key
  key = LAST_KEY
  IF key = 27 THEN EXIT get_key
  IF key = 9 THEN SCREEN_DUMP
END REPeat get_key
STOP
```

7.16 NET_STATION

A useful little routine if you use the QL network a lot. If you have forgotten the number of the station you are working on, simply use this function to remind you.

7.17 BAUDRATE

Another useful little function, this time to remind you of the current baudrate set on your machine.

7.18 FIND_TEXT_IN_ROM

A routine to anable you to find any text in the ROM. Try looking for command names or keywords to see where they live in the ROM, or even look for names in the ROM, e.g. look for Sinclair. The routine uses the SEARCH_I function to look for a string which will match if in upper case or not. You can adapt this routine to become a general purpose routine to search within any given area of memory.

7.19 LINE_INPUT$

This example provides a function similar to INPUT, but allows text to be presented for editing and also allows ESC to be pressed to end the routine.

There is a list of 6 parameters to this routine, as follows.

"channel" is the screen channel to use, e.g. #1
"aty" is the y co-ordinate to be used, values as in the AT command
"atx" as "aty", but x co-ordinate across
"ending" is the code of the character which ended the input
codes currently supported are 208 and 216 for cursor up
and down keys, 10 for ENTER and 27 for ESC. Note that it
is a returned code, not one supplied to the routine,
though the calling variable should have been pre-defined
to avoid an error (any dummy value will do). The returned
value allows the program to make a decision on next
action depending on how this one was finished (e.g. to
stop if the entry was terminated with ESC or to move up
or down a list if the cursor up/down keys were used.
"maxlen" is the longest length of entry permitted, e.g. 20
characters.
"prompt$"is the text supplied to the routine for editing, e.g. a
default drive name.

You could call this routine with a line such as:

```
LET how_ended = 0 : LET drive$ = 'MDV1_'
LET a$ = LINE_INPUT$(#1,5,5,how_ended,40,drive$)
SELect ON how_ended
  =27 : PRINT "ESC"
  =10 : PRINT "ENTER"
  =208: PRINT "Cursor up"
  =216: PRINT "Cursor down"
END SELect
```

Here, we are entering a string in channel #1, 5 characters across the window and 5 down. It is to be a string no more than 40 characters across and the default is 'MDV1_'. You can edit it on the screen using the normal cursor left and right keys and typing whatever text is required, as you would in an INPUT statement. When you have finished, press ENTER to finish normally, or ESC to quit. You can also quit by pressing cursor up or down if the entry was part of a list to be altered.

Some other keypresses are supported, such as ALT left and right to get to the ends of the string being edited, CTRL left and right for deleting characters while editing and CTRL ALT left or right to delete the whole line while editing.

This routine is a bit slow on a standard unexpanded QL, but fast enough when compiled.

7.20 FETCH_BYTES_JM$

Due to a problem with the input buffer on a QL with a version AH or JM ROM, the FETCH_BYTES function will stop with a buffer full error if you try to fetch more than 128 bytes from a file. This is also true for GET_STRING, see below. The cure, quite simply, is to split it up to fetch 128 bytes at a time and this routine shows you how to do this. The routine checks if the QL has a version AH or JM QL by using VER$, then if it has, it splits up the string fetched into 128 byte chunks. This is much slower than the usual FETCH_BYTES function, but at least it works on a version AH or JM QL.

7.21 GET_STRING_JM$

This routine is similar to FETCH_BYTES but performs a similar action for the GET_STRING command which can fail for the same reason as the FETCH_BYTES function.

Strings fetched with GET_STRING have a two byte length specifier followed by the characters of the string itself. So GET_WORD is used to fetch this two byte part first, then the FETCH_BYTES_JM$ function is called to fetch the rest of the string in 128 byte chunks.

Note that both routines test the ROM version and use the normal (faster) commands if not a version AH or JM system.

When fetching strings from files and writing strings to files, you should always be careful with odd length strings. In some cases, the operating system ensures that string lengths are even by adding an extra dummy character at the end (which is normally invisible) if the string itself has an odd length.

7.22 FIND_FILE

This routine, though written as a single procedure, is a complete, complex program in its own right. Its purpose is simple - given a drive name and a short text string (less than a kilobyte long, though the routine is easily adapted for longer ones if required), FIND_FILE will hunt through files on a disk or cartridge, scanning them for the string. For example, if you want to find letters to a Mr. Jones, simply put the disk you wish to search in a drive and type in

```
FIND_FILE 'FLP1_', 'Jones'
```

It does not matter if you use upper or lower case letters and you can even use directories if you wish. FIND_FILE shows the filenames of files containing the string on the screen and a short extract from the file either side of where the string was found (about 140 bytes in all). Up to 20 filenames can be listed before the program pauses and asks you to press a key to continue.

While this program is useful in its own right, there are some changes you can make to improve it. The best thing you can do is buy Norman Dunbar's The Gopher program from DJC (which is far better than this effort and has more facilities) to get some ideas of what else is possible.

This program fetches about 2 kilobytes at a time from the file, overlapping by enough text to cover the length of the string being searched for. If you wish to make it read larger chunks at a time, change the value of the variable called "buffer" a few lines from the start of the FIND_FILE buffer. While increasing the value of this variable will speed up searches, it also needs extra memory to hold the buffer string.

Other things which can be added:

a. Make the routine search through all directories it finds, you will need to add an extra procedure to scan through all files, picking out the sub-directories with the FILE_TYPE function and calling this routine for each one found to be type 255 (directory). You will, of course, need to modify FIND_FILE itself to ignore directory files and you may need some clever code to handle sub-directories within directories!

b. Add a printout option. This has been marked with a REMark statement in the middle of the procedure.

c. Add an option to only search certain file types (e.g. DATA files or JOB files)

d. Add an option to only search files with given filename endings

e. Add an option to only search, say, 10 kilobytes into the file to speed up searches.

7.23 YN$

A simple, short routine which waits for you to press Y for Yes or N for No. It enables a cursor while it waits for the keypress and converts upper case letters to lower case, so it always returns a y or an n.

7.24 DISPLAY_SIZES

This is a function which helps you to detect what type of display is in use. This function uses error trapping to open known maximum sizes of display for various hardware available. It returns a number from 0 to 3, representing the type of display:

3 is a QVME card (from Jochen Merz), the maximum display size tested for is 800 x 300.

2 is Extended Mode 4 on the Atari ST-QL emulator, resolution 768 x 268.

1 is the mono mode on the Atari ST-QL emulator, resolution 640 x 400.

0 is a standard QL resolution display, 512 x 256

-1 means that the routine was not able to test the display sizes and abandoned without a result (-1 is the QDOS error code for 'Not Complete').

The routine currently tests for the Toolkit 2 FOPEN function or the QLiberator Q_ERR_ON error trapping keyword. If neither is found (tested with CHECK) the program cannot function. But a REMark statement shows where you could add a WHEN ERRor statement as error trapping as the

third option for the routine if your ROM version supports WHEN ERRor (version JS, MG, or Minerva, for example).

I am grateful to Ralf Rekoendt (Germany) for supplying the information on display sizes, without which this routine would not be possible.

7.25 IS_POINTER_ENVIRONMENT

This routine checks if pointer environment is present and prints up a message to say if it is installed or not. You can use similar routines in your programs to allow the use of keyboard control if the pointer environment is not present. One example of a program which can be pointer driven, or can be controlled from keyboard if the pointer environment is not installed, is QLiberator version 3.

7.26 REPORT_ON_FONTS

The address of the pair of fonts for the channel number specified is printed, along with the details of the lowest valid character code and number of characters in the font. You can use this information to copy a character set from ROM into RAM, for example, and redefine a few characters as required. This routine then goes on to print the characters (not including the default character) of both fonts.

7.27 LIST_HEAP

If you have made multiple common heap allocations and perhaps CLEARed out the variables holding the addresses, you may well find that the QL starts to run short of memory, because of blocks of memory are still present as reserved blocks. When the variables holding the addresses of these blocks have been cleared, it is difficult to clear out the heap. This procedure shows how to list the blocks allocated by DJToolkit, so that you can see how many there are and remove them with RELEASE_HEAP if desired. It picks up the address of the heap from the system variables and steps through all entries it finds. It checks that it is a DJToolkit allocation (as distinct from Toolkit 2, Turbo Toolkit, MegaToolkit, etc allocation) by checking for the "NDhp" string at the start address - 4 of the block (DJToolkit actually allocates 4 bytes more than asked for and uses those four to hold this little identifier header, or "signature"). NOTE, the Toolkit 2 routine CLCHP will NOT clear out any of DJToolkit's reserved areas.

This procedure has been updated to reserve a couple of areas in the heap so that you can see something on the screen when it is called. It does NOT release these areas, so now you also have a valid reason to call the next routine CLEAR_DJTK_HEAP don't you ? (N. Dunbar 22/10/93)

7.28 CLEAR_DJTK_HEAP

Why clear them out manually when the computer can do it for you? This is an adaptation of the LIST_HEAP procedure, which stores the addresses in the array addrs(). Up to one hundred addresses can be handled, which should be quite enough! Note that the blocks are released in reverse order to help avoid heap fragmentation. Please note that if heap allocations have been made using extensions from other toolkits, this routine will NOT clear them out, you may well find you have a fragmented heap afterwards as well in that case - there is no substitute for being tidy and deallocating heap memory after you have finished with it. You can use CLCHP from Toolkit 2 to clear out its heap allocations, which may help.

7.29 DISPLAY_WIDTH_JM

As mentioned in the manual, the DISPLAY_WIDTH function can not always return reliable values on a version JM or AH QL - the manual hints at using VER$ to test for these QL ROM versions, but does not explain how. Here is the solution, and this routine is used by a few others in this demos file. It simply returns the 'fixed' value of 128 bytes per line if it is a version AH or JM QL which cannot have a screen of different size anyway. PLEASE NOTE: A fix for this problem has been built into the DJToolkit from version 1.11 onwards so this routine is now not required, though it will be included for a while for use with older versions of the toolkit.

7.30 ERROR_MESSAGE$

This is not a demonstration file for the DJToolkit as such, more a short routine to turn the error codes returned by some functions into a meaningful message like the ones the QL gives when something goes wrong. It accepts the error code as its parameter and returns a string with the message. It is used by some of the example routines.

7.31 LARGE_TEXT

This routine is a simple character enlargement routine which prints text by using the letters of the string to be enlarged to represent the pixels of each character. This is obviously of limited use on the screen, but you may be able to adapt it to make similar enlarged characters for printing on paper (perhaps several letters per pixel for large characters for banner-making). It is a simple way of enlarging characters and saves messing about with bit image graphics if you intend to print large characters to a printer! The parameters for the routine are: channel number, y and x co-ordinates down and across the channel concerned, and finally the text string to be printed.

7.32 LOAD_A_FONT

Shows how to load a font for a given channel by checking its length (which also serves as a rudimentary error trap by checking the value returned by FILE_LENGTH), reserving the required amount of memory in the common heap, using USE_FONT to assign the character font to the channel specified, print a sample string to the channel using the new font and finally resetting to default font before releasing the heap memory and ending. If you wish to experiment with this routine, there is a small number of fonts on the disk. Fonts have filenames ending with _FNT to make them easy to spot in a directory listing. Note that this routine assumes that the font file only contains one font - some font files contain the fonts for set number 1 and 2 of each channel, the second one is ignored.

7.33 NARROW TEXT

This routine needs the NARROW_FNT font file on the disk to be loaded (it looks for it and reserves memory itself). It shows how to use SET_XINC and SET_YINC to squeeze more characters on the screen. To demonstrate, it acts as a typewriter. Simply type enough characters (a few lines) of text to see what it looks like, press ESC to quit. You should get over 120 characters per line with the 4 pixel increment set by this routine, though unless you have a good monitor it won't be particularly readable. Note that if the window in which such reduced text is displayed has a border, it may be damaged by some x increment values (give this window a BORDER#0,1,255, for example and see what happens when the cursor reaches the right of the screen).

7.34 FASTCOPY

This uses the DJToolkit heap and file handling commands to set up as much of the common heap as practical for fas copying by loading the files into RAM and saving as many in one go as possible. On the Gold Card this will be no faster than using the WCOPY command in many cases, but can make quite a difference on older interfaces. The routine also sets the file type in copied files, so is not just limited to data file copying. It is quite a long procedure and difficult to follow unless you are used to long, complex BASIC routines! It requires two parameters, the drive to copy from and the drive to copy to.

7.35 BACKUP_BY_DATE

The file header in QL files can contain three important dates, the file update date (when a file was last changed), the file reference date (not used in many systems) and the file backup date (when a backup was last made, but this is not implemented on most QL systems). This routine compares the update date with the backup date and makes a backup copy if required (i.e. it has changed since the last backup). This means that if you have a disk full of files and only a few have changed, only those few files need to be copied, not the whole lot, thus giving you a quick and simple automatic backup facility. There is, however, a snag (isn't there always?). A normal QL doesn't set the backup date when you copy a file, for example, so we have to check if the command SET_FBKDT is present (it's a command built into the Gold Card, and on Miracle's hard disk system) to be able to set the backup date after making a copy.

If it is present, life is sweet again. The routine wades through all the files it can find on the disk or directory specified, compares the update date with the backup date and if the file has been updated since the last backup date, copies the file and changes the original file's backup date. It does not change the backup date on the copy of the file, unless you make another backup of that too!

7.36 CLEAR_BACKUP_DATES

Where the previous routine comes unstuck is when something happens to the backup copy, or you wish to make another backup from the source disk. Since the date has been changed, you are stuck, catch 22. However, it is possible to get around this by clearing the backup date by setting it to a DATE value of 1 (0 is not used, because it is ignored by the Gold Card version of the SET_FBKDT command), effectively back to the start of 1961!

At this point, a plug for a piece of DJC software! If you think these last two routines are clever, wait until you try Norman Dunbar's Winback backup utility which is even cleverer than these routines. Backup by dates, by directory, splitting long files, and much more is possible with this super program - if you have ED drives or HD drives on your Gold Card system, you can use Winback to make backups of those onto cheaper DSDD disks. If you have a Miracle hard disk, Winback is one of the most useful add-ons you can have for it, for only £25.00 (and I can heartily recommend it for all users ! (N. Dunbar 22/10/93))

7.37 ALTER_DATASPACE

A short routine which allows you to inspect the dataspace of an executable program and modify it. Often, no useful purpose is served by this, and in some cases it can be an unwise thing to do. But if a compiled program runs out of memory due to insufficient dataspace (this can sometimes happen with Turbo compiled programs if they were given the wrong dataspace to start with), this program can help you to increase the dataspace. Simply enter the name of the file to be inspected, then it shows the current dataspace and prompts you to enter a new value. To leave it unchanged, enter the same value.

8. DJTOOLKIT BASIC DEMONSTRATION ROUTINES 2

by Dilwyn Jones, June 1994

The files DEMOS2_bas and DEMOS2_sav (QSAVEd version) contain a further set of demo routines to show how to use the new commands and functions available in DJToolkit from V1.15. Basically, these are the fill memory, file open, float peek/poke and MAX_ extensions.

Many of these routines use quite complex bits of code, so do not be discouraged if you find them difficult to follow at first if you do not have much experience of basic programming.

Several of these routines use MAX_CON. This uses the iop.flim trap which returns the screen size if testing a primary channel (usually lowest CON channel, e.g. #0 in Superbasic), or the maximum size within the outline of the primary channel - the outline is the outer limits of a window in pointer environment terms (there is a bit more to it than that, but the full explanation is too complex to list here). This means that other window numbers cannot be lager in size than this outline size.

For superbasic, applying MAX_CON to #0 will generally tell you the maximum screen size provided that the iop.flim trap is implemented on your system. If you have pointer environment installed, for example, it should be present, though MAX_CON may not work on early QL rom versions.

If in doubt that MAX_CON is returning the full screen size rather than the outline size, check the result given by MAX_CON against that given by DISPLAY_WIDTH.

There is a routine in DEMOS1_bas to determine maximum window size by error trapped trial and error if you wish to do it that way (i.e. using WINDOW to size down from large window size until it comes into range, error trapping the command for out of range errors).

8.1 SCREEN_SIZES

Shows how to use MAX_CON to return the maximum possible display size (see pointer environment warning above), allowing for the fact that the trap may not be implemented on some early QL systems by setting default values for the screen dimensions to those of the standard QL video screen (512 x 256 pixels). It returns the width, height and origin values for the standard three superbasic windows. For #0, it will return 512,256,0,0. For the other channels it depends on the outline set for the basic windows. In general, if you don't know about outlining, it won't affect you and you do not need to know about it to use this demo routine.

8.2 CLEAR_WHOLE_SCREEN2

Fills the display with 16 bit values (0) to clear the whole screen to black, by using FILLMEM_W. See warning re. pointer environment above, which is why it tests channel #0 rather than #1 or #2. This routine is broadly equivalent to WINDOW #0,512,256,0,0 : CLS #0 but writes directly into the display area. Though this is not good practice, it is nonetheless possible and shows how to do this in a reasonably safe way.

8.3 PEEK_FLOAT_DEMO

Sets up a table of 10 six byte floating point numbers in memory by using RESERVE_HEAP to make room for the table, then using POKE_FLOAT to store the ten random numbers. It then asks you to enter a number from 1 to 10 for the number to be recovered from the table. If you enter a 0, the routine comes to an end.

8.4 L2_DIR

This routine only works with Gold Card, Super Gold Card, hard disk or other device (such as emulators) which supports real directories created with a command such as MAKE_DIR, with file type 255 (i.e. not Thor directories).

This is a complex routine which allows you to get a list of all files in a given directory on a given drive, including files living in sub-directories. This routine requires two parameters, the drive name and the directory name. If you wish to look at the root directory (i.e. see every single filename on that drive), specify the subdirectory as a null string (L2_DIR "win1_","") - on hard disk systems, for example, this can result in a very long list of files. The routine calls itself recursively every time it finds a subdirectory name until it runs out of names. Note how each variable needs to be a local variable for this routine to work, so each time the routine calls itself it can create a new set of variables each time and remember them for the previous call when it returns to that. The routine lists filenames and directory names (directory names followed by -> like the DIR command).

The DJ_OPEN_DIR function is used to access the file headers in the directories studied on the drive. Each entry is 64 bytes long and fetched with FETCH_BYTES. The name is extracted from this along with the file type byte. If this is 255, the name under study is considered to be a sub directory and the routine calls itself again to read the contents of that directory and so on. Because the action is basically the same for each subdirectory read and there is a finite return point each time (i.e. a definite end of the list of files in a directory) this is an ideal example of a routine which calls itself recursively. Recursion is a mind-bending subject to those who have never studied it before, so if you want to know more about it, find a good programming textbook (or ask Norman Dunbar). Recursive routines can use up a lot of memory, which may be an important consideration when compiling a program containing routines such as this one.

This routine may be useful to anyone who is considering writing software which needs to access and manipulate files in sub directories, such as backup programs, file searchers, or listing utilities.

8.5 DIR_ARRAY

This routine is based on the previous one, but rather than just producing a listing of files in a given directory, this routine returns a list in an array so that your program can access the filenames if required. It makes two passes through the list of files, the first is only to check for lengths of names and how many names need to be fitted into the arrays. This information is used to dimension two sets of arrays, one containing subdirectory names, the other containing the filenames. The array filenames$() contains the filenames, while the array called dirs$() contains the directory names. The four variables indicated at the beginning of the procedure called DIR_ARRAY with REMark statements contain details such as how many filenames, how many directory names, length of the longest names etc.

8.6 SAVE_BIG_SCREEN

On hardware such as the Miracle Systems Ltd QXL which allow the use of screen sizes larger than the standard QL 512x256 screen, it is useful to have a method of being able to save the screen automatically without having to ask the user what width and height to save. The QL manual describes how to save the current screen picture with the command:

```
SBYTES 'filename',131072,32768
```

This is fine for the standard QL screen, but for the larger screens of the QXL and enhanced mode 4 on the Atari ST-QL emulator, for example, a better method is needed as several factors are likely to change:
(i) The screen base address may not be at address decimal 131072 (hex 20000).
(ii) The width of the screen may be different (e.g. 640 or 800 pixels on the QXL, 768 on the enhanced mode 4 of the Atari emulator).
(iii) The height of the screen may also differ.

This routine first of all works out the size of the screen in pixels using MAX_CON applied to the primary basic channel (#0), then working out the number of bytes per line with DISPLAY_WIDTH. As the QL screen is a series of lines of pixel, all of the same length, it is a simple matter to work out the length of the area to save by multiplying the width of one line in bytes by the height of the screen in lines. The SCREEN_BASE function is also used to find the address in memory at which the screen starts. I have also used the primary channel for SCREEN_BASE and DISPLAY_WIDTH, as there is the possibility (as advised in the Minerva manual) that future display hardware could theoretically have different windows in different copies of the screen where the hardware and operating system permitted this.

8.7 STORE_BIG_SCREEN

Rather than store the screen as a file on disk, this function allocates some common heap memory (with RESERVE_HEAP) and uses DISPLAY_WIDTH, SCREEN_BASE and MAX_CON to check the screen size and allocate enough memory accordingly. The function returns either the base address of the area of memory used to store the screen address, or a negative number which will be an error code. MOVE_MEM is used to transfer a copy of the screen to the area of memory reserved. If you really must directly address the screen rather than use operating system calls, e.g. for speed reasons or because no operating system facility exists, these routines are reasonably safe ways of doing so and correctly used should ensure your program works on future QL hardware.

8.8 RESTORE_BIG_SCREEN

This routine restores a picture from the reserved area of memory directly onto the screen. Just as before, the DISPLAY_WIDTH, SCREEN_BASE and MAX_CON functions are used to check screen dimensions and MOVE_MEM is used to transfer the picture back to the screen. RELEASE_HEAP removes the area of common heap memory used, and the variable holding the common heap address is zeroed so that the calling routine knows the common heap area used has been released.

8.9 SAVE_PTR_SCREEN

Requires one parameter - the filename of the file to be saved. This routine saves a copy of the screen picture in a file format known as the area save bitmap format used by many pointer environment programs for graphical applications. Line Design is one well known example of a program which can use this format. It is a standard format for saving whole screens or areas of the screen in such a way that it can be loaded into any application which supports this format. The files have a few bytes of preamble at the beginning to identify them as this type of file.

2 bytes Hex'4AFC' Decimal 74,252
1 word width in pixels
1 word height in pixels
1 word line increment in bytes (number of bytes between
        the start of one line and the start of the next)
1 byte screen mode (usually 4 or 8)
1 byte spare (not used, reserved for future application)
followed by the bit image data in the same form as that of the screen picture memory itself.

First of all, MAX_CON is used to check the size of the current screen. Then DISPLAY_WIDTH is used to check the width of each line of the display in bytes. Note that for standard mode 4 and mode 8 displays, the line width in bytes will always be INT(pixel_width/4), but this is a dangerous assumption to make as any future graphics hardware may well implement modes which do not conform to this rule. But for standard QL mode 4 and 8 displays, it is a useful rule of thumb.

Next, the DJ_OPEN_NEW function is used to try to open a file with the given filename. If the file already exists, DJ_OPEN_NEW will return the error code -8 (which corresponds to the QL error message 'already exists'), which is a useful way of implementing a 'OVERWRITE YES/NO?' type of option for saving files, as we can act on the '-8' value obtained. The user is asked to state by pressing Y for Yes or N for No if he/she wishes to delete the existing file with that filename. If N for No is pressed, the routine comes to an end.

If Y for Yes is pressed, DJ_OPEN_OVER deletes the file and has another go at opening a new file for us.

PUT_WORD and BUT_BYTE are used to build up the file preamble for us (10 bytes in all). SCREEN_BASE tells us where to start saving the screen from, then PEEK_STRING is used to fetch each line of the display from memory, with the number of lines being kept in the variable h% from the earlier MAX_CON statement. PRINT is used to send each line to the file before the file is finally closed.

8.10 LOAD_PTR_SCREEN

Requires one parameter, the filename of the screen to be loaded.

This routine loads a picture previously saved in the area save bitmap format with the routine described above onto the screen. The area loaded may be smaller than the screen itself - this would allow a standard QL screen picture to be merged onto a larger QXL or Atari screen, for example.

First of all, MAX_CON and DISPLAY_WIDTH are used to check the display size details. DJ_OPEN_IN is then used to try to open the picture file - if it fails, the routine simply returns without error, other than a low pitched beep which I use to signify errors of some sort.

FILE_LENGTH is used to check the length of the file. Since the area save bitmap files include a ten byte preamble in the file at the very least, files which are less than 10 bytes long can be ignored. This routine also shows an approach to decoding files of certain types, an approach you may be able to use if your program needs to be able to manipulate certain types of files only. It is common practice to place a few bytes at the start of the file as markers to identify files, and it only takes a very small amount of time for a program to open the file and check these first two or four bytes, which drastically cuts down the risk of causing all sorts of problems by accidentally loading another program's files. First of all, FETCH_BYTES fetches two bytes from the file to check for the hex $4AFC flag at the beginning which identify the area save bitmap files. GET_WORD and GET_BYTES are then used to fetch the width, height and mode details. As there is one spare, unused byte in the file, MOVE_POSITION is used to skip over it.

Next, the file dimensions are checked against those of the screen, in case the picture is too big to be loaded onto the screen. The mode of the picture is checked against the screen mode and changed if necessary. Note how the routine checks the mode, changes it, and checks again to see if the mode change was successful. Usually, this will not be necessary, but some versions of the Atari ST-QL emulator did not have MODE 8 built in, so attempting to change to mode 8 would not be successful.

If anything went wrong, the file is closed and the procedure returns.

To load the picture itself, we check the base address of the screen and fetch one line at a time from the file onto the screen display directly. The 'a' loop ensures that the correct number of lines are fetched. Note the warning in the listing for owners of version AH or JM QL roms (you can check your rom version with PRINT VER$), which may give buffer overflow errors if FETCH_BYTES is used to fetch more than 128 bytes at a time from a file due to a limitation in those rom versions. There is a FETCH_BYTES_JM$ routine in DEMOS1_bas which can be used in its place if this causes you a problem.

POKE_STRING is used to place the string of bytes corresponding to one line of the picture direct into the display.

8.11 OPEN_OVER_QUERY

Requires one parameter - the filename of the file to be opened.

Tries to open a channel to a file, and if that file already exists, prompts you to choose whether or not to delete the file. The error code returned by DJ_OPEN_NEW (in the variable 'chan') is checked for a value which indicates that the file already exists, and prompts to ask if the file is to be deleted or not.

This routine allows you to build routines into programs which act rather like the Toolkit 2 'Already exists, overwrite Y/N?' prompts, without having to rely on the recipient of your programs having Toolkit 2 on his/her system.

8.12 TK2_DATA$

This routine looks in the system variables and finds the Toolkit 2 data default drive (as set with the DATA_USE command in basic). If the default is not found, the function returns a null string, otherwise it uses a rather complex PEEK_STRING statement to recover the default drive details.

Although it is easy enough to use the Toolkit 2 DATAD$ function to find the default drive of course, this routine allows you to check if the default itself is present, so your program can take alternative action if Toolkit 2 is not present. The data default drive is the drive from which data files, screens, and even basic programs are loaded from by default, though see under TK2_PROG$ below for details of basic programs loaded and saved using QLOAD and QSAVE.

8.13 TK2_PROG$

This function returns the Toolkit 2 program default drive details, in much the same way as the TK2_DATAD$ function above. The program default drive is the one programs are executed from, or that which Liberation Software's QLOAD and QSAVE utilities use to load and save basic programs from.

8.14 TK2_DEST$

A function which returns the current Toolkit 2 default destination name, as set with DEST_USE.

8.15 OPEN_IN_TK2

Requires one filename, that which the function is to open. Normally, DJ_OPEN_IN etc functions do not pick up the Toolkit 2 default drives, but if you want them to do this and act like the Toolkit 2 redefined file opened commands, this function shows how to do this, but also means that your program can also work on systems without Toolkit 2.

8.16 FILE_BROWSER

A useful little routine which allows you to list files on all of your drives with just one or two keypresses. It shows the device names on your system (FLP, MDV, RAM, WIN, DEV etc) which are obtained by using the DEV_NAME function, and allows you to choose drives 1 to 8 for those device names (often there will only be one or two floppy drives, for example, although up to 8 ramdisk drives amy be present on most systems).

To select a drive, press the first letter of its name (e.g. F for FLP). To select drive number, press a number from 1 to 8 (e.g. 1 for FLP1_). Sub directories are scanned using the L2_DIR routine described earlier.

9. DJ TOOLKIT UPDATES LIST

9.1 UPDATES TO DJTOOLKIT V1.10

A few new commands have been added, such as QPTR, the font handling commands and DISPLAY_WIDTH to check the display size. All these are now documented in the manual. Norman has, however, given me an embarrassing list of typing errors I made when I transferred the manual to prepare it in Text87, these will be corrected in the next issue of the manual, as they do not affect the accuracy of the manual, only offend those who dislike typos!

9.2 UPDATES TO DJTOOLKIT V1.11 (18.5.93)

Despite the fact that the AH and JM ROM presented the DISPLAY_WIDTH command with problems, this has now been solved in two ways. Firstly, a demo routine (DISPLAY_WIDTH_JM) checks for an offending ROM version and returns a default value to prevent the problem. Secondly, Norman has patched the DISPLAY_WIDTH function to include a check for AH and JM ROMs (or rather the versions of QDOS with those versions of BASIC, to be accurate) to prevent the problem.

I have fixed an embarrassing number of faults in the demo files. Nobody actually complained about these, I just noticed them myself. I've also added a few more demo routines such as a fast copier and dates utility - most of the new routines are at the end of the file.

Note that the DEMOS_sav version can give a list of missing extensions when loaded with QLOAD if the QLiberator extensions are not present. For the most part, they will still run OK, since a check is made for their presence (e.g. the CURSOR_ENABLE routine. I have also updated the DEMOS_doc documentation file to include details of the new routines.

I made a few changes to the demo routines to take account of the fact that Norman reprogrammed some functions in V1.10 at the suggestion of Ralf Rekoendt of Germany, to return negative error codes rather than stopping with an error message.

The first commercial program using this toolkit was launched - DJC's CONVERT-PCX graphics conversion utility, for clipart ported from the PC (used in conjunction with Discover). CONVERT-PCX costs just £10.00.

9.3 UPDATES TO DJTOOLKIT V1.12 (15.6.93)

Due to the fact that I tried to make things a bit quicker in the MOVE_MEM command, I ended up using an algorithm that allowed an easy (!) way to figure out which direction the memory needed to be moved in order to avoid overlap problems. As it turned out, the algorithm was wrong ! This caused a slight problem in that some of the first 6 bytes were not moved when moving from an even address to an even address with no overlap, the program did it as if there was an overlap and missed a few bytes out of the move.

MOVE_MEM is now fixed, bigger and for small memory moves, it spends most of its time figuring out how to actually do it. Large memory moves, say saving and restoring screens, should now work correctly and quickly.

9.4 UPDATES TO DJTOOLKIT V1.13 (19/07/93)

I use a Gold Card for all my work, it is quick and the vast amount of memory allows me to run lots of utility programs together with QPAC 2 etc. So what, I hear you think. Well it seems that a small bug has existed in the FILE_POSITION function which causes a normal 128K QL to crash with a fancy screen display which fills the screen from the bottom to the top - interesting. A Trump Card (old version, no level 2 drivers) just gives up quietly and gives no indication of its troubles. A Gold Card works !!!!!!!

It seems that the system call FS_POSRE (TRAP #3, D0 = $43) actually destroys register A1 which is of course the maths stack pointer. This has now been fixed for All QLs, not just the Gold Card users. Funny, no one has complained about it up until yesterday when Dilwyn Phoned !

Having tested the new version (1.13) on a Trump Card equipped QL, I fired up the trusty Gold Card and traced the execution of FILE_POSITION using QMON 2. Lo and behold, the A1 register is PRESERVED by the system call FS_POSRE (and FS_POSAB ?) when running with a Gold Card, mystery solved, but why is it preserved ?

9.5 UPDATES TO VERSION 1.13 PART 2 (22/10/93)

Dilwyn contacted me to say that a customer was having problems with some of Dilwyn's demo routines. I have had a look at these (Dilwyn has more than enough problems with Page Designer 3 !!!!) and found that most of them were caused by not having enough LOCal statements.

Some of Dilwyn's routines use the same names, but some are ARRAYs and others are not. If FASTCOPY has been called, LOAD_A_FONT refuses to work due to the variable 'fl' being a DIMmed array in FASTCOPY (for file length). I have added a few more LOCals to every routine that needs them. Problem now solved.

You should be aware that on some QLs, JS in particular, there is a bug that occurs when a program routine is executed. If the routine (PROC or FN) has a total of 10 or more parameters and locals then the SuperBasic listing gets trashed in a big way. The program will probably fall over with BAD

NAME or something.

When testing the amended demo routines, I of course had forgotten about this bug and managed to remove the SuperBasic job from the QL all together (who said it couldn't be done ?) Using the JOBS/RJOB utilities in QPAC 2 did not even show SuperBasic as a job any more !!!!!

Luckily I always (?) save changes before running them, just in case. One quick reset later and all was well again. Enough waffle, hopefully the demos are now ok. I have put a warning in the demos file at the start and REMarked out extra LOCal lines but there shouldn't be any more name clashes - famous last words.

9.6 UPDATES TO DJTOOLKIT V1.14 (12/06/94)

At Dilwyn's request, some additional routines have been added to the toolkit. These being some file opening functions that return an error code or the channel id. I also added a couple of extra handy routines of my own, just for fun. The new routines are :

```
POKE_FLOAT address,value      (PROC)
PEEK_FLOAT(address)           (FN returning float)
MAX_DEVS                      (FN returning integer)
DJ_OPEN('filename')           (FN returning integer)
DJ_OPEN_IN('filename')        (ditto)
DJ_OPEN_NEW('filename')       (ditto)
DJ_OPEN_OVER('filename')      (ditto)
DJ_OPEN_DIR('filename')       (ditto)
MAX_CON(#ch, x,y,xo,yo)       (FN returning int + altered params)
```

The file opening procedures are very siimilar to Simon Goodwin's recent article in the DIY Toolkit series in QL WORLD magazine (Vol 2, issue 8 which was marked Vol 2 issue 7 just to be confusing). The article was about his routines called ANYOPEN%. Simon's article came in very handy as I had known about the ability to extend the SuperBasic channel table, but had not quite figured out how to fill it in afterwards. Thanks Simon.

9.7 UPDATES TO DJTOOLKIT V1.15 (16/06/94)

So, I thought it was complete, but Dilwyn left a message on my machine, which went something like "what happened to the fill memory commands then ?" - oops, I forgot !

This version, 1.15, now contains the additional procedures :-

```
FILLMEM_B start_address, how_many, value
FILLMEM_W start_address, how_many, value
FILLMEM_L start_address, how_many, value
```

and that is about what they do !