qb64forum.alephc.xyz

MEM Blocks by Luke

Author

15–19 minutes

MEM Blocks
Author: @luke
Last updated 2016-08-22

A _MEM block is a predefined UDT, with the following elements (all are read-only):

Element	Data type	Purpose
OFFSET	_OFFSET	Memory address of start of block.
SIZE	_OFFSET	Size of block in BYTES.
TYPE	LONG	Bit-flags describing type of data.
ELEMENTSIZE	_OFFSET	Size of datum in bytes.
IMAGE	LONG	Image handle (if appropriate).

_MEM.OFFSET

A pointer to the beginning of the data in memory. Doing maths with this is always in bytes (_MEM.OFFSET + 1 is always the second byte in the block, regardless of the type of data). Can be passed to DECLARE LIBRARY routine if the routine's parameter is BYVAL p%& and the routine expects an int32_t (which is really more of a void*; cast it to the appropriate pointer type). C routines can then treat it as an array.

_MEM.SIZE

The size of the block in BYTES, not the number of data. Could possibly be 0. Note the data type is an _OFFSET not a LONG (or similar), so QB64 may have issues with how it is used (for instance, you can't assign it to a LONG).

_MEM.TYPE

Note:

- Bits are numbered starting at 0, which is the least significant bit.
- Exclusive means nothing else is set (except for Array. Array goes with anything).

Bit	Meaning w.r.t datum	Combinability (if it be set along with other flags)
0	1 byte large.	
1	2 bytes large.	
2	4 bytes large.	
3	8 bytes large.	
4	16 bytes large (unused).	
5	32 bytes large.	Byte sizes are only set for simple numeric
6	64 bytes large (unused).	types, including _OFFSET, and for pixel data.
7	Integral type.	
8	Floating-point type.	7-9 set whenever appropriate, including pixel data.
9	String type.	Exclusive.
10	Unsigned type.	Goes with integral types and pixel data.
11	Pixel data from image.	Sets byte size, integral and unsigned.
12	_MEM U.D.T	Exclusive.
13	_OFFSET data type.	Goes with integral and byte sizes.
14	Created by _MEMNEW or _MEM(x, y)	Exclusive.
15	U.D.T other than _MEM.	Exclusive.
16	Array.	Other flags describe type of array element.

_MEM.ELEMENTSIZE

The size of each datum in the block, in bytes. If the block was created with _MEMNEW or _MEM(x, y), this value is 1. If _MEM.TYPE has the Array flag set, this is the size of one array element.

_MEM.IMAGE

If the block was created by the _MEMIMAGE function, this is the handle of the image that was used. Otherwise, it is -1.

Command for creating and manipulating _MEM blocks.

General Notes

- When a function requires 'block' and 'offset', 'offset' must be at least 'block.OFFSET', and at most 'block.OFFSET + block.SIZE - 1'. Of course, if the function accesses multiple bytes, the upper limit on 'offset' is decreased.
- Some functions accept a parameter of type DTYPE. This is the literal word to refer to a data type, such as INTEGER or _UNSIGNED LONG.
- When referring to an entire array (not just an element), empty parentheses must be used.
- An array reference with a specific element "x(3)" is interpreted as a single variable, except for the one-argument form of the _MEM() function.
- Multidimensional arrays are stored (0, 0), (1, 0), (2, 0) ... (9, 0), (0, 1), (1, 1), (2, 1) etc.
- Elements in a UDT are simply stored one after the other.

_MEM

In the first form, creates a new _MEM block referring the data in 'var', where 'var' is a numeric type, fixed-length string, UDT or array of such types. If an array is specified with an element e.g. "x(3)", the function takes this to mean an array beginning x(3) and all elements above that. It is vitally important to understand that the new _MEM block does not copy data from 'var'; the memory region is the same as 'var'. 'var' and the _MEM block are now two ways of accessing the same part of the computer's memory. This means that if 'var' is changed (by assignment, not with _MEM commands), the value in the _MEM block will change too. Similarly, using _MEMPUT to change the _MEM block will change the value of 'var'. As you may expect, if two _MEM regions 'm1' and 'm2' are both created with this function form from the same 'var', altering one will alter the other. If 'var' no longer exists (for instance, the SUB it existed in has finished), then the block is considered to have been freed, accessing the _MEM block is an error.

In the second form, creates a _MEM block to access memory beginning at 'address' in the computer's memory, and 'size' bytes long. Unlike _MEMNEW, which allocates a block of the size requested, _MEM() in the second form assumes that the memory at 'address' has already been allocated. This form is most useful when a function through DECLARE LIBRARY returns the address and size of a buffer it has stored information in; in this situation, the two data can be passed to _MEM() so that the _MEM commands can be used to access the data.

However, this second form gives great freedom, to the extent that it cannot always catch errors. If 'address' is incorrect, or 'size' is too large, it is possible to write to memory that the program is not allowed to access. This will generate a segmentation fault, which will either cause the program to crash immediately or trigger an OS-level error message (and then crash). If you're not using DECLARE LIBRARY, there's a good chance that you will never need this second 'unsafe' form of the _MEM function.

_MEMELEMENT

---

block AS _MEM = _MEMELEMENT(var AS ANY)

Like the one-argument form of the _MEM() function, creates a _MEM block which can be used to access 'var'. Unlike _MEM() though, if 'var' is an array with an element specified e.g. "x(3)", it is interpreted as a single variable only. Note that the Array flag of _MEM.TYPE is still set, even though the _MEM block only contains one datum. Other than that, _MEM.TYPE and _MEM.ELEMENTSIZE are set as one would expect for a single variable. Changing the data in the _MEM block will alter the element in the original array.

_MEMNEW

Returns a a _MEM block that refers to newly allocated memory where 'size' is its size in bytes, and may be 0. The contents of the block are unspecified; if a definite value is needed, _MEMFILL can be used to initalise the region. The allocation is not guaranteed; the underlying libaries and Operating System may fail to allocate the requested size. It is always prudent to verify that that it was successful by comparing 'block.SIZE' to 'size' - an inequality means failure. If it does fail, the program may try again with a smaller size, or give an error to the user then exit. Failed allocations must be freed with _MEMFREE. If 'size' is negative an Illegal Function Call is raised, but the returned block still needs to be freed. All _MEMNEW allocations have 'block.ELEMENTSIZE' set to 1.

[Author's note: I was able to successfully allocate a block up to the maximum size on my 32 bit machine (2^32/2 - 1) thanks to the magic of virtual memory, where the Operating System does not actually allocate physical RAM until it is used. I suspect I would not be able to actually assign data to all of it, given I only have 4 GiB of RAM installed.]

_MEMIMAGE

Note: the 'handle' argument is optional. If omitted, it defaults to the current write page (which by default is the one being displayed).

Returns a _MEM block that references the memory holding the image referred to by 'handle' (where 'handle' is a handle as returned by _NEWIMAGE and related functions). This can be seen a counterpart to the _MEM(x) function, in that the memory accessible IS the image; changes to the _MEM block affect the image itself. If the image is the active screen, changes are seen immediately (assuming _AUTODISPLAY). 'block.IMAGE' is set to 'handle'. If 'handle' does not refer to a valid image or refers to a hardware surface (image mode 33), Illegal Function Call is raised, and the returned _MEM block does not need to be freed. If the image is freed with _FREEIMAGE, the _MEM block is freed too.

For all graphical screen modes, the memory is a pixel-by-pixel representation of the screen. Each row of pixels on screen is placed one after the other, with the top row first. As an example, consider a 5x5 screen, with pixels labelled like so:

```
ABCDE
FGHIJ
KLMNO
PQRST
UVWXY
```

In memory, these pixels would be in alphabetical order. However, a pixel is not necessarily a byte. In 32 bit mode, each pixel is 4 bytes wide: one pixel per colour channel, in the order Blue, Green, Red, Alpha. Correspondingly, 'block.ELEMENTSIZE' is 4 for 32 bit mode. The programmer should be careful with the order of colour channel components, especially due to the reversing nature of little endian processors. To this end, the _RGBA32() function and the related colour functions should be used instead of working with raw numbers.

In text mode (SCREEN 0), we talk of character cells instead of pixels. Nevertheless, the translation for screen location to memory location is the same as it is for graphical pixels. This is in contrast to the usual column, row method of addressing text modes. Each character cell is two bytes (as recorded by 'block.ELEMENTSIZE'): the first cell is code-point of the character being displayed, the second stores the attribute information as set by COLOR. See the manual on the COLOR statement for more detail about the format of attribute information.

_MEMGET

_MEMGET block AS _MEM, offset AS _OFFSET, dest AS ANY
dest AS ANY = _MEMGET(block AS _MEM, offset AS _OFFSET, type AS DTYPE)

Accesses data in the _MEM block specified by 'block' at the address 'offset'. In the first form, the type of the data is inferred from the type of 'dest'. In the second form, the type is explicitly stated. Multibyte data types are considered to have their first byte at the location specified. On little endian machines at least, numeric types are read natively i.e. with the least significant byte first. This function is an excellent way to access the bit-by-bit representation of complex data types such as floating point numbers.

_MEMPUT

Stores the data 'src' in the _MEM block specified by 'block' at the address 'offset'. Arrays, UDT's and strings (both variable and fixed length) are allowed. For numeric literals, it is necessary to use the second form to explicitly state the type of a variable, since it would be ambiguous otherwise (is "8" 1 byte wide, 2 bytes or 4 bytes, or maybe even a SINGLE?).

_MEMFILL

_MEMFILL block AS _MEM, offset AS _OFFSET, size AS _BYTE, src AS ANY
_MEMFILL block AS _MEM, offset AS _OFFSET, size AS _BYTE, src AS type AS DTYPE

Like _MEMPUT, stores the data 'src' in the _MEM block specified by 'block' at the address 'offset'. However, _MEMFILL then

repeats this storage as many times as necessary to fill a region of memory 'size' bytes large. As for _MEMPUT, the second form is used when the type of a numeric literal needs to be explicitly stated. It is important to realise that since 'size' is a number of bytes, it is possible to specify a multibyte data type that does not fill the region exactly. For instance, a LONG (4 bytes) will fill a 10 byte region with 2 instances, plus 2 remaining bytes. In this case, _MEMFILL will use the first 2 bytes of the LONG to finish filling the region, despite this resulting in an incomplete representation of the data being stored in the last instance.

_MEMCOPY

Note: The "TO" is literal, and must be included in the statement.

Copies 'size' byte of data from 'srcoffset' in 'srcblock' to 'destoffset' in 'destblock'. 'srcblock' and 'destblock' may be the same. The source and destination regions are allowed to overlap each other; in this case, the copy will be done so as to do the Right Thing. _MEMCOPY makes a proper copy of the data. Unlike _MEM(x) and _MEMIMAGE, which simply create a reference to a location in memory, _MEMCOPY duplicates each byte. Altering the source at a later time will not alter the destination copy. 'size' may be zero (in which case no copy is performed) but may not be negative.

_MEMEXISTS

Returns -1 (true) or 0 (false) to indicate if 'block' is a valid _MEM block or not. A block is considered valid if it refers to a _MEM block, and has not been freed. See _MEMFREE for discussion of when a _MEM block is considered freed. Library routines in particular should be prudent about verifying a _MEM block's validity before accessing it.

_MEMFREE

Frees the memory region associated with 'block'. 'block' is now considered invalid, and any attempts to use it (other than creating a new block) are an error. If any other references to the memory exist, such as a variable when _MEM(x)/_MEMELEMENT() is used or an image when _MEMIMAGE is used, the memory may still be accessed through them without error. Attempting to free a _MEM block that has already been freed (or was never valid) is an error. Note that a _MEM block referring to a variable or image is considered freed if the variable or image no longer exists.

It is possible to leak memory if all references to a _MEM block created with _MEMNEW or _MEM(x, y) are lost (such as going out of scope when returning from a SUB) before they are feed. For this reason, the programmer should be careful to track all _MEM blocks and free when necessary. It is not necessary to free _MEM blocks immediately before a program exits.

$CHECKING

Enables and disables runtime safety checks, particularly for _MEM commands. As a meta-command, it has effect regardless of whether it is in some kind of conditional statement (since it is parsed at compile-time). $CHECKING:ON is the default, but turning it off will give a significant speed boost for code that uses _MEM commands. Checking can be turned off for only a section of code by surrounding it with $CHECKING:OFF ... $CHECKING:ON. When checking is off, what would have given a runtime error will now cause a segmentation fault, or simply overwrite parts of memory. Despite the dangers, once code is tested and working well it is well-worth turning off checking for small parts, especially inner loops.
****

Data type equivalence table

C type	QB name	QB Symbol

N/A	_UNSIGNED _BIT	~`
N/A	_BIT	`
N/A	_BIT * n	`n
N/A	_UNSIGNED _BIT * n	~`n
int8_t	_BYTE	%%
uint8_t	_UNSIGNED _BYTE	~%%
int16_t	INTEGER	%
uint16_t	_UNSIGNED INTEGER	~%
int32_t	LONG	&
uint32_t	_UNSIGNED LONG	~&
int64_t	_INTEGER64	&&
uint64_t	_UNSIGNED _INTEGER64	~&&
float	SINGLE	!
double	DOUBLE	#
long double	_FLOAT	##
qbs*	STRING	$
qbs*	STRING * n	$n
ptrszint	_OFFSET	%&