Qualcomm Technologies International, Ltd.

VM Memory Mapping and Memory Usage

Application Note

80-CT415-1 Rev. AJ

October 27, 2017

Revision history

Revision	Date	Description
1	NOV 2008	Initial release. Alternative document number CS-00110364-AN.
2	OCT 2009	Updated to latest style guidelines
3	JUL 2010	BlueLab references removed and minor editorial changes
4	AUG 2011	Updated to latest CSR™ style
5	JAN 2012	Updates to sections 2 and 3
6	JUN 2013	Updated to latest CSR style
7	APR 2014	Updated to latest CSR style
8	SEP 2016	VM memory external SRAM region updated. Updated to conform to QTI standards.
AJ	OCT 2017	Added to the Content Management System. Updated DRN to use Agile number. No change to technical content.

Contents

Tables

Figures

1 BlueCore's CPU processor

Qualcomm® BlueCore™ technology's CPU has the following physical and architectural properties:

- BlueCore's CPU processor has 16-bit registers.
- Data fetches are 16-bit wide.
- The CPU has a Harvard architecture, which maximizes efficiency from the small program and data memory available on the chip, by keeping code out of the data address space.
- The firmware controls the chip memory and allocates requested memory to the VM in blocks if it is available.
- The VM controls the applications view of the RAM available to it.
- The VM prevents corruption of sensitive registers or firmware data structures.

2 VM memory map

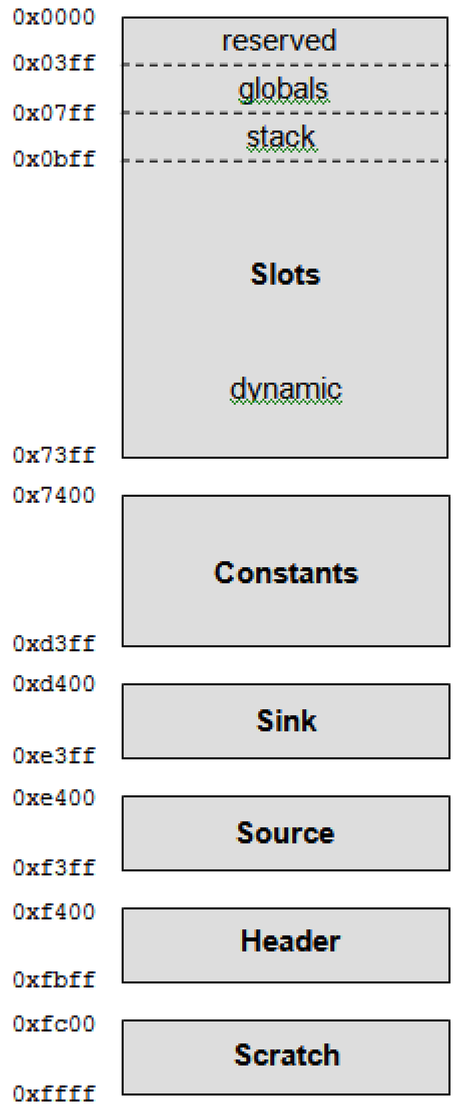

Figure 2-1 VM memory map

2.1 VM memory Slots region

Addresses `0x0000` to `0x3fff` are mapped to 64 slots that start on 256 word boundaries, that is:

`0x0000`, `0x0100`, `0x0200`, and so on.

Table 2-1 lists slots that are reserved for special use.

Table 2-1 Slots reserved for special use

Address	Description
`0x0000 - 0x00ff`:	This slot is reserved in order to trap any NULL pointers. It does not contain any data. Therefore, if an application attempts to access memory from this slot by dereferencing a NULL pointer this is recognized as an error and the VM panics the application.
`0x0100 - 0x04ff`:	These slots are reserved for the application stack.
`0x0500 - 0x08ff`:	These slots are reserved for storing the global variables.

The other 54 slots are available for use as dynamic memory. However, one slot must always be available to allow messages with a payload to be handled. If no slot is available, then the VM will panic the application.

NOTE The memory window in xIDE can be used to monitor the number of slots being dynamically allocated during runtime.

When the application requests a new block of dynamic memory (by calling `malloc`, `PanicUnlessMalloc, PanicUnlessNew`) the VM attempts to claim a memory block from the Firmware. The Firmware then attempts to allocate a block of at least the size requested by the `malloc` call. If the Firmware is successful the VM finds an unused slot and maps the memory block to it.

If the allocation requested size is more than the 256 word boundary, then the Firmware attempts to find a multi-slot region to accommodate the size request. The Firmware will only be successful if the unused multi-slots are physically adjacent such that, the complete allocation can be easily accessed by the application without any fragmentation using the base address returned to the application.

While the size of the block may be larger than requested the VM knows the size requested and does not allow attempts to write or read data outside the requested memory. If an attempt is made to access an address outside the requested memory the VM panics the application.

2.2 VM memory external SRAM region

The SRAM region is used to map external SRAM into memory. The application is able to read or modify the external SRAM contents when the application has mapped the required external RAM region using the `SramMap()` function.

From unified-28b firmware, a new feature has been added that allows SRAM to be used for allocating memory for the VM applications. The SRAM region can be accessed by the application to allocate memory using `SramAlloc()`. These allocations share the slots used by `malloc()`. Total memory allocation, from both internal memory and SRAM, is limited by the number of heap slots. Memory allocated using `SramAlloc()` can be freed using `free()`. Existing memory operations `realloc()`, `memmove()`, `memset()`, and `memcpy()` can be used to update the memory allocated by `SramAlloc()`.

2.3 VM memory Constants region

The Constants region maps to ROM/Flash memory used to store constant data that is required to persist for the duration of the program.

The Constants region is limited to a maximum of 24 K words and is used for:

- Constant global variables
- String literals e.g. “Hello world”
- Jump tables for switch statements
- Initializers for non-constant global variables

The linker produces an error message if the 24 K limit is exceeded.

2.4 VM memory’s Sink, Source and Header regions

The sink, source and header regions are used to map streams into memory. Only one stream can be mapped into each region at any one time. Subsequent calls to `SinkMap()`, `SourceMap()` or `SourceMapHeader()` invalidate the pointers received from the previous call.

2.5 VM memory Scratch region

This is reserved for QTIL engineers.

3 C source code and BlueCore memory

Considering the limited memory available to VM applications it is important to have an understanding of how variables in the C source code are stored/handled in memory.

Global variables

For example, `int a = 1;`

Variables declared outside functions persist throughout the program and are stored in the globals region. If (as in the example above) they are initialized, the initialization value is stored in the constants region.

Global constants

For example, `const int b = 2;`

Variables declared as global constants persist throughout the program and are stored in the constants region.

Local variables and parameters

For example, `int foo (int x, int y, int z)`

```
{
const int r=x*y*z;
static int c;
return r+a+b+c++;
   }
```

Local variables and parameters are held in the stack or in registers and only exist till the function returns. Thus in the example `int x`, `int y`, `int z` and `int r` are held on the stack or in registers.

NOTE The compiler decides where each variable is stored while the function executes.

Although the variable `int c` is declared within the function `foo` it is marked static. Therefore, its value is persistent between function calls and it is stored in the globals region.

3.1 Purpose of memory regions

The table describes the memory regions and their runtime access limits.

Table 3-1 Memory region runtime access limits

Region	Address	Purpose	Maximum Available	Runtime Access Limit	Writable
Slots	`0x0000`	Unused	n/a	n/a	n/a
	`0x0100`	Stack	Chip/firmware dependent	As set by application at link time	Yes
	`0x0500`	Globals	Chip/firmware dependent	As set by application at link time	Yes
	`0x0900` to `0x3f00`	RAM	54 slots (size of each slot is allocated by firmware and is chip/firmware and current application usage dependent)	As allocated by application and calls to `malloc`[2]	Yes
SRAM	`0x6400`	Mapped external RAM	4 K words	As returned by `SramMap()`	Yes
Constants	`0x7400`	Storage of Constants Global variable initializers, string literals, jump tables and initializers	24 K words	Set by application at link time	No
Sink	`0xd400`	Mapped sink	Sink type and chip/firmware dependent	As returned by `SinkClaim()`	Yes
Source	`0xe400`	Mapped source	See note [1]	As returned by `SourceSize()`	No
Header	`0xf400`	Mapped header	Source type dependent	As returned by `SourceSizeHeader()`	No
Scratch	`0xfc00`	n/a	n/a	n/a	n/a

NOTE [1] Dependent on source mapped. Typically, if a file is mapped in, then the first 3 K is visible. If the UART is mapped in then up to 0.5 K of data may be visible depending on the amount of data in the UART receive buffer.

[2] `malloc` calls depend on available RAM and the corresponding `pmalloc` pool configuration in the firmware, see *Configuring the Qualcomm BlueCore Memory Allocator*.

3.2 Internal and external fragmentation of VM memory

Applications that allocate and free memory dynamically may experience shortage of memory due to fragmentation problems. Fragmentation of available memory addresses can occur as a result of external or internal fragmentation.

External fragmentation

When applications allocate memory in certain specific sequences, it is possible that the applications may not be able to allocate more memory even though the required memory is available due to external fragmentation.

For example, an application that attempts to allocate 19 x 1-slots and 18 x 2-slots alternately, creates:

[1], [2,3], [4], [5,6], [7], [8,9], [10], [11,12], [13], [14,15], [16], [17,18], [19], [20,21], [22], [23,24], [25], [26,27], [28], [29,30], [31], [32,33], [34], [35,36], [37], [38,39], [40], [41,42], [43], [44,45], [46], [47,48], [49], [50,51], [52] [53, 54], [55]

If all the 1-slot allocations are then freed and the application requests another 2-slot allocation, the request is denied, even though a total of 19 slots (non-consecutive slots) are unused/available.

Therefore, such requests result in external fragmentation issues and should be avoided where possible.

To minimize losses due to external fragmentation:

- Allocate large/larger blocks of memory before smaller blocks in the application code.

  NOTE Allocation of big blocks for global data or stack is handled automatically by the firmware which designers can take advantage of when writing their program(s).

Internal fragmentation

A single memory slot contains up to 256 words. The firmware cannot allocate an address space lower than 256 words, this means that small allocations (or allocations which are not exact multiples of 256) waste address space.

For example, a single memory allocation for 257 words requires the VM to reserve 2 slots, i.e. 514 words. This effectively wastes 255 words of addressable memory.

4 BlueCore memory protection

The VM is designed to protect memory used by the firmware. This prevents the application code corrupting memory data critical to the underlying functionality of the BlueCore IC.

In addition the VM panics if the application attempts to:

- Read outside a region: In the case of RAM slots this is the size of memory requested not necessarily the actual size of memory block mapped to the slot.
- Write to a read-only region, i.e:
  - Constants region
  - Mapped source or source header
- Access a region that does not exist, i.e:
  - A slot that has not been allocated a memory block by the firmware
  - An unmapped source

The panic is fatal. Depending on the panic action setting in the project properties, the application panics but the firmware continues to run or the firmware panics as well and the chip requires a reset.

NOTE If the application is being run in xIDE only the application will panic whatever the Panic action in the project properties is set to. This allows xIDE to maintain a connection to the chip for display and debugging purposes.

Certain traps exposed by the firmware may have restrictions on the destinations of pointers that are passed into them.

For example, pointers into mapped streams passed into traps may cause the VM to panic. Consult the VM and Native library reference guide of the SDK documentation for detailed information.

NOTE Valid pointers into RAM (i.e. the slots region) may always be passed into traps, so it is always possible to copy data to/from a RAM buffer in order to pass it to traps.

5 Frequently asked questions about VM

Q1. What happens when VM requests dynamic memory e.g. malloc is called?

Answer:

The VM finds an unused slot. If there are no unused slots the `malloc` fails. If there is an unused slot the VM requests a memory block from the firmware, if a large enough memory block is not available the `malloc` fails.
If the firmware has a large enough memory block available its address is given to the VM and the VM binds the block to the slot and marks the slot as active. At this point the memory is available to the application.
Even if the memory block bound to the slot is actually larger than the amount of memory requested only the requested memory is valid.
If there is an attempt to access more than the requested memory the VM panics.

NOTE All other allocation functions behave in the same way.

Q2: What happens when the application code releases memory i.e. free is called?

Answer:

The VM checks the pointer. It must point to the start of a slot that is bound to allocated memory. If either of these checks fail the VM panics the application. If the checks do not fail the slot is marked as unused and the memory block is returned to the control of the firmware.

Q3: How is memory allocated and used by messages?

Answer:

The application must request the memory required to hold the message structure of the message being sent. Typically this is done using `PanicUnlessMalloc`, e.g:

```
/* request memory size of CL_XXXX_CFM_T */
CL_XXXX_CFM_T *message = PanicUnlessMalloc(CL_XXXX_CFM_T);
/* fill in payload */
message -> messagefield1 = data1;
message -> messagefield2 = data2;
/* send the message */
MessageSend (getAppTask(),CL_XXXX_CFM_T, message);
```

If the required memory is available a memory block is bound to a slot as described in question 1. If the memory is not available the VM panics the application.

NOTE If the engineer does not want to evoke a panic, `PanicUnlessMalloc` can be replaced with `malloc`. The code must then check the result before filling the payload. If `malloc` fails, it is the engineer’s responsibility to decide whether to free some memory, wait until some memory may be available before trying to send the message again or not to send the message at all.

When `MessageSend` is called the memory is queued for delivery and the slot is marked as unused.

When the message arrives at the front of the queue the message subsystem looks for a free slot, if one is available the memory block is bound to it to it and tells the recipient task there is a message at that location. The VM panics the application if no unused slot can be found for message delivery.

When the task reads the message the slot is freed and the memory returned to the firmware. Thus undelivered messages tie up memory blocks not slots.

Q4: How are firmware primitives handled in the VM?

Answer:

Primitives from the firmware can be considered to behave very like messages. However, the connection library has been developed to provide an interface to the firmware and it is inadvisable for designers to use any other method of communicating with the firmware.

Q5: How is the memory required by globals and the stack allocated?

Answer:

The memory required for globals and the stack is calculated by the compiler and allocated when the application starts running. If either is too large the application will not run and an error message will be generated by xIDE at this time.

Q6: Can I set the stack size manually?

Answer:

Yes. The size for the stack can be set manually using the Project Properties in xIDE. However, as stated in question 5 the tool chain can usually calculate the optimum size for the stack based on the C Source code.

The exception is if function pointers or recursion is used in the code, either of which means that the tool chain cannot accurately calculate the stack size and the stack must be set manually. However, as getting the size of the stack wrong can be fatal to the application Qualcomm recommends the use of function pointers and recursion is avoided so that the size of the stack can be automatically calculated and set.

Q7: My application has too many global variables?

Answer:

If the globals in an application exceed the maximum allowed the code needs to be rewritten so that less globals are used. This can be done by defining them locally when they are needed or allocating a memory slot to store some variables when the program loads.

The `<project_name>.map` file generated by xIDE when the application is built, is a useful resource that identifies all the globals in the application and where they are used.

Q8: The stack required to run my application is too large?

Answer:

If the application requires a stack that is too large to run on the chip, the functions which require the most local variables to be stored on the stack when called, must be rewritten to reduce the number of variables they call locally.

The `<project_name>.map` file generated by xIDE when the application is built, is a useful resource that identifies which functions in the application are using the most words in the stack when called.

Document references

Document	Reference
Configuring the BlueCore Memory Allocator	80-CT406-1/CS-00128085-AN

Terms and definitions

Term	Definition
BlueCore	Group term for the range of QTIL Bluetooth wireless technology ICs
Bluetooth	Set of technologies providing audio and data transfer over short-range radio connections
CPU	Central Processing Unit
IC	Integrated Circuit
QTIL	Qualcomm Technologies International, Ltd.
RAM	Random Access Memory
ROM	Read Only Memory
SRAM	Static Random Access Memory
UART	Universal Asynchronous Receiver Transmitter
VM	Virtual Machine