Zalt Memory Management

## Hardware

The Memory Management Unit hardware consists of a 32k static RAM (15ns) which data lines output the extended memory address bus MA12-MA19. The lower 4 address lines of the RAM (A0-A3) are driven by the upper 4 address lines of the Z80 CPU A12-A15. The other RAM address lines are used as follows:

* A0-A3 driven by CPU A12-A15  
  This is a memory map table of 16 bytes
* A4-A11 are driven by the table selector latch.  
  An output instruction writes a new value to the latch. There are two table selector latches. One for normal operation and a separate one for targeting the table to configure.
* A12 switches between read tables and write tables.  
  In normal operation this is connected to the CPU /RD signal
* A13 is a spare (0)
* A14 is reserved for use by the System Controller.  
  This allows the system controller to setup a complete set of separate memory mappings.

A memory-map table consists of 16 bytes. A table byte represents the memory address lines MA12-MA19. 256 memory-map tables can be stored and switched between quickly (1 output). There can be 256 separate memory-map tables for read and 256 separate memory-map tables for write.

The boot default memory address for MA12-MA19 is xFF due to pullup resistors. The System Logic (CPLD) maps this to MBE0 (default 64k memory bank). To target MBE1 (the second 64k memory bank) use value x00, for MBE2 (3rd 64k bank) use x01 etc.

Initializing a memory-map table requires on output instruction to select the table (latch) and one output instruction to write the data to the table itself.

## Memory Scenarios

The following memory usage scenarios are considered and used to validate the design:

1. Allocate large objects
2. Segmented code / transfer execution
3. Local vars / Stack
4. Bank Switching
5. Share read-only/executable memory (code) between tasks
6. Checked pointer usage
7. API / tracking blocks
8. CRT integration

Calling between pages (transfer):

* Call a proxy from the client that brings in the correct memory block and calls the routine.
  + How are routines identified?   
    RST is probably easiest. => requires an \_init at every bank? Not when bios page is fixed.
  + How are arguments passed?   
    Separate args via registers or stack.   
    Could also enforce use of structs then only required to pass one ptr.
  + Is a common stack required?   
    Yes => routines are called ‘in process’.
  + Is it possible to bring a block in into any free space (routine in question must be relative)?
* Transfer should be a common API for reuse.
  + Extensible RST handler (16bits: libid-fnid)
  + Static init of jump table (least overhead).

Allocate large data:

* Statically
  + Built into c-compile process. Named memory\_section (sdcc) or far ptr support.
  + Need program header to describe program parts. Memory segments is one of this.
* Dynamically
  + API similar to malloc but with VirtualAlloc semantics.  
    Allocation is a reservation, handle is returned. Opening a specific space/length will commit memory blocks and map into address space => where? Anywhere, because its data and ptr is returned from handle.   
    Or specify fixed flag and fixate address – needed if data contains other addresses (ptrs) within the same block.  
    Close will unlock (when not fixed) and allow remapping in the future.
  + Based on allocation flags (fixed) targeted on different heaps.
* Heap  
  A pool of memory resource to be allocated dynamically
  + Tied to one or more memory pages.
  + Either all fixed or all relocatable (based on underlying pages)

Local vars and stack:

* Sdcc supports named address regions. Which will solve local vars.
* Stack must stay fixed. Need process/thread struct to maintain memory table and for future multi-tasking.

Memory Bank switching:

* Use one current mem-map table (per thread) and mutate that table as needed (1 out instr.).  
  This would give us max 256 concurrent tasks (good).  
  Alternatively a table per physical bank per thread…
* Using multiple tables for one thread introduces the complexity of how to decide which one to take (1 out instr.) Performance wise it doesn’t matter.

Checked pointer usage/memory access:

* Access through API  
  Z80 has no intrinsic facilities.
  + long AllocMem(flags)
  + void\* LockMem(handle, flags)
  + bool CheckAccess(ptr, flags)
  + void UnlockMem(handle/ptr)
  + void FreeMem(handle)
* Error handling?  
  When Carry flag is set on return of sub routine, this indicates an error.
* Read/Write (data), read-only, write-only?, execute (read-only), copy-on-write?
* Data, static (bss), code, user, resource (read-only data)

Block usage admin - Zalt-os (saltos ;-):

* Mem-mgr struct
  + mem-banks  
    How many (and which) memory banks are loaded?
  + mem-pages struct  
    What memory pages are there?
    - mem-page struct
      * Page Id (not stored, can be calculated from bank and mem-map table values)
      * Type and Access flags (data/code/bss, read/write/execute)
      * Relocation flags (fixed)
      * System flags (stack, heap etc)
* Thread struct  
  Id  
  Register storage for context switch.  
  Memory Management struct.
  + Mem-Map Table Index (8 bits)  
    The number of the selected mem-map table when this thread is executing.  
    Assigned when program is loaded and thread is started.
  + (ptrs to) 16 active mem-page structs for active mem.
    - Access flags (read/write/execute)
    - Relocation flags (fixed)
    - Ptr to next block (for block ranges)
  + (ptrs to) Offline mem-page structs for allowed access/usage (how many? dynamic?)
* What system pages need to be fixed?
  + - \_init  
      Implements RST and NMI end points. Fixed at x0000.
    - stack  
      Grows down.
    - heap  
      Grows up.
    - interrupt vector table  
      Fixed length = x0100
    - bios  
      Functions need to be available.
  + Fixed System Pages:
    - Page 0 (and up? => depends on how much bios code needs to stay resident)  
      \_init, bios
    - Page x (can be located anywhere but is fixed)  
      heap, stack interrupt vector table per thread-swapped on thread-switch

P15: 4k

P14: 4k

P13: 4k

P12: 4k

.  
.  
.

P3: 4k

P2: 4k

P1: 4k

P0: 4k

x0000

Max: Bank 15

xFFFF

Bank 0

\_init

x0100