TN202

# Rabbit Memory Management In a Nutshell

The Rabbit CPU has a Memory Management Unit (MMU) that controls how logical memory addresses map into physical addresses, and a Memory Interface Unit that controls how physical addresses map into actual hardware.

The Dynamic C compiler and libraries generally handle memory mapping details so that most Dynamic C users don't have to concern themselves with it, but some advanced applications may need to manipulate the MMU and/or the MIU.

For further details on memory management, see the Rabbit microprocessor user's manual for your Rabbit chip.

## Definitions

**Physical (or Linear) Addresses** - 20-bit addresses representing the 1M address space that logical addresses map into. The highest 20-bit address is 0xFFFFF. The default addresses in the Dynamic C development system flash start at physical address 0x00000 and in RAM start at physical address 0x80000.

**Logical Addresses** - 16-bit addresses representing a 64K address space. The highest 16-bit address is 0x0FFFF. Most Rabbit instructions use logical addresses. The location in physical memory where these addresses map into is completely controllable by the programmer via the MMU.

**Segment** - A block of logical memory; the block sizes are multiples of 4K (0x01000)

**Bank** - A 256K block of PM, on a 256K boundary (0x40000). There are four banks available within the Rabbit physical address space. The starting address of each bank is 0x00000, 0x40000, 0x80000, and 0xC0000.

## Memory Mapping Unit

The MMU translates a 16-bit logical address to a 20-bit physical address.

The logical address space is divided into four segments: *xmem*, *stack*, *data*, and *base*. The xmem segment always occupies E000h–FFFFh. The other segments are located from 0000h to DFFFh, and are adjustable in size to anywhere from 0 bytes to E000h bytes. The total size of all three is always E000h. The stack segment starts above the data segment and always ends at DFFFh. The data segment is always above the base segment. The boundaries between the base segment and the data segment, and between the data segment and the stack segment are set by an MMU register, SEGSIZE. The upper nibble of SEGSIZE represents the stack-data boundary (in 1000h byte units), and the lower nibble represents the data-base boundary (in 1000h byte units).

Each of the upper three segments has an associated segment register used to map logical addresses that fall within the segment to the physical address space. From top to bottom, they are:

- XPC
- STACKSEG
- DATASEG.

In each case, the physical address is computed by shifting the segment register left 12 bits (multiplying by 1000h) and adding it to the l6-bit logical address. Here is the algorithm for converting logical addresses to physical addresses.

### Abbreviations

**LM** - Logical Memory
**PM** - Physical Memory
**LA** - Logical Address - address within the LM
**PA** - Physical Address - address within the PM

Let SEGSIZE = XYh where X is the high nibble and Y is the low nibble.

```
If LA >= E000h
    PA = LA + (XPC x 1000h)
Else If LA >= X000h
    PA = LA + (STACKSEG x 1000h)
Else If LA >= Y000h
    PA = LA + (DATASEG x 1000h)
Else PA = LA
```

Any of the segment registers can be loaded with a new valid value at any time, but this has to be done with great care. For example, if code is executing in the *xmem* segment and the XPC is changed, then execution will not continue at the next instruction, but instead will continue at the location in physical memory where the logical address of the next instruction maps to. This is because the PC (program counter) register holds a logical address.

## Example

The MMU registers are set as follows:

```
XPC        =  0xF8
SEGSIZE    =  0xD6
STACKSEG   =  0x92
DATASEG    =  0x7A
```

The physical address of the bottom of the xmem segment is given by:

`0xF8000 + 0x0E000 = 0x06000` (bit 20, the 21st or carry bit, is ignored)

The physical address of the bottom of the stack segment is given by:

`0x92000 + 0x0D000 = 0x9F000`

The physical address of the bottom of the data segment is given by:

`0x7A000 + 0x06000 = 0x80000`

Note that the data segment starts at the beginning of the "normal" RAM space and, for a 128K RAM, the stack segment is allocated 4K.

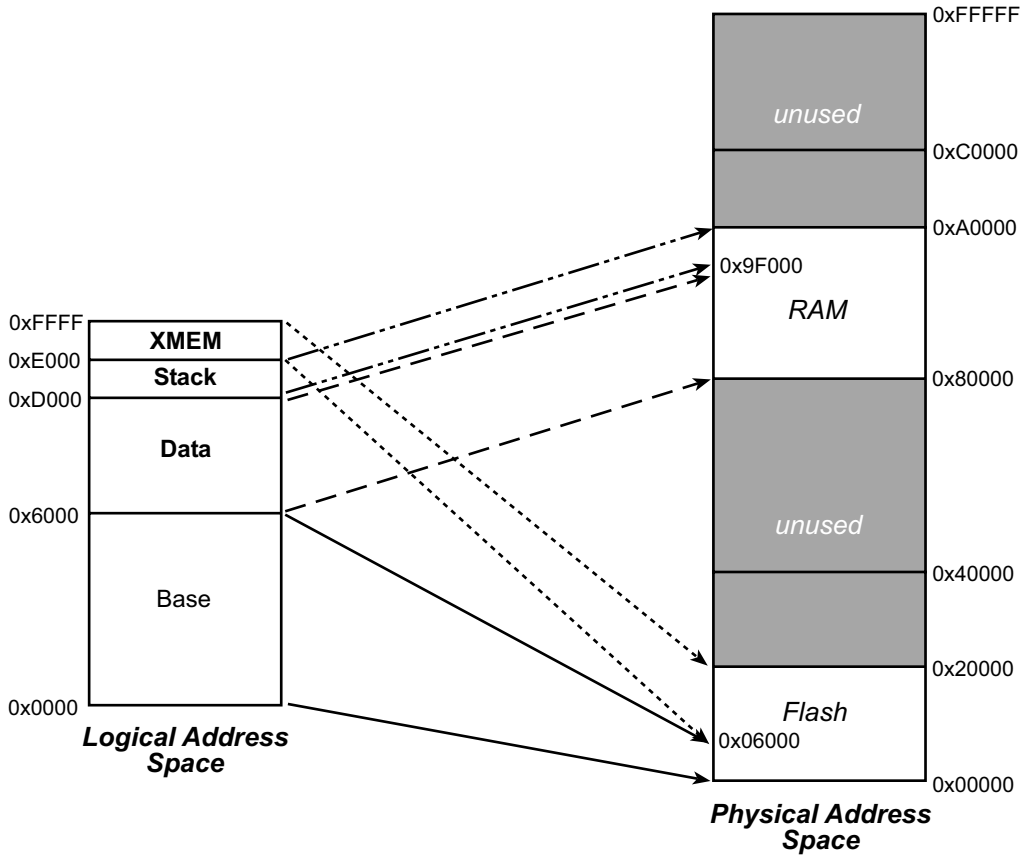
There are special Dynamic C functions to access data at a specific PA: `root2xmem()` and `xmem2root()`. There are also several assembly language instructions:

- `LDP` for memory access
- `LCALL` and `LJP` for branching

Note that the LDP instructions bypass the MMU and go directly to the MIU (see the next section).

The following diagram illustrates the memory mapping discussed above using the sample calculations with a 128K flash and a 128K RAM.

**Figure 1. Memory Mapping Between Logical Address and Physical Address**




## Memory Interface Unit

The MIU controls memory access after the MMU determines the physical address.

There are five registers associated with the MIU:

- MMIDR
- MB0CR
- MB1CR
- MB2CR
- MB3CR

The primary function of the MMIDR register is to allow the system to permanently enable /CS1. This could allow faster access to the RAM by keeping the chip always selected. However, you will need to insure that the /OE and/or /WE signals are not shared with another device.

Each of the four Memory Bank Control Registers MB0CR, MB1CR, MB2CR and MB3CR. controls one 256K quadrant of the 1M physical address space. These registers control wait states, CS/WE/OE line usage and write protection.

The MBxCR registers also can be set up to invert address lines A18 and A19. If bit 4 of one of these registers is set, the MIU inverts A18 after the logical to physical conversion is done by the MMU. If bit 5 of one of these registers is set, the MIU inverts A19 after the logical to physical conversion is done by the MMU.

For a 256K or smaller device, address line inversion has no effect since the highest address line needed to address 256k is A17, but for a 512K device, this means that we have access to the whole 512K via one 256K quadrant of physical memory. Because we can control A19 as well, each 256K quadrant of physical addresses can actually address a whole 1M. This, in combination with different CS/WE/OE combinations controllable by the same registers, gives the Rabbit access to 6M of memory with no external glue logic.

### Example

MB2CR is set to map to CS1/WE1/OE1, and these lines go into a 512K RAM. Therefore physical addresses 80000h-BFFFFh (quadrant 2 of 0-3) map to offsets in the chip of 00000h-3FFFFh (the lower 256K of the chip) using lines A0-A17 with A18 not asserted. Now we set bit 4 of MB2CR to invert A18. Physical addresses 80000h-BFFFFh now map to offsets in the chip of 40000h-7FFFFh (the upper 256K of the chip.) using lines A0-A17 with A18 asserted .

## MMU/MIU Registers

### Segment and Memory Bank Registers

Table 1 lists the details about the segment registers.

**Table 1. Segment Registers**

| Register Name | Mnemonic | I/O Address | R/W | Post Reset |
|---|---|---|---|---|
| MMU Instruction/Data Register | MMIDR | 10h | R/W | xxx00000 |
| Stack Segment Register | STACKSEG | 11h | R/W | 00000000 |
| | Locates stack segment in physical memory | | | |
| Data Segment Register | DATASEG | 12h | R/W | 00000000 |
| | Locates data segment in physical memory | | | |
| Segment Size Register | SEGSIZE | 13h | R/W | 11111111 |
| | Bits 7..4–boundary address stack segment<br>Bits 3..0–boundary address data segment | | | |

Table 2 lists the details about the memory bank control registers.

**Table 2. Memory Bank Control Registers**

| Register Name | Mnemonic | I/O Address | R/W | Post Reset |
|---|---|---|---|---|
| Memory Bank 0 Control Register | MB0CR | 14h | W | 00000000 |
| Memory Bank 1 Control Register | MB1CR | 15h | W | xxxxxxxx |
| Memory Bank 2 Control Register | MB2CR | 16h | W | xxxxxxxx |
| Memory Bank 3 Control Register | MB3CR | 17h | W | xxxxxxxx |

## Memory Bank Control Register Functions

This table details the functionality of the memory bank control registers and the necessary bit values.

**Table 3. Memory Bank Control Register x (MBxCR=14h+x)**

| Bit(s) | Value | Description |
|---|---|---|
| 7:6 | 00 | 4 wait states |
| | 01 | 2 wait states |
| | 10 | 1 wait states |
| | 11 | 0 wait states |
| 5 | 1 | Invert address A19 |
| 4 | 1 | Invert address A18 |
| 3 | 1 | Write-protect memory this quadrant |
| 2 | 0 | Use /OE0, /WE0 |
| | 1 | Use /OE1, /WE1 |
| 1:0 | 00 | Use /CS0 |
| | 01 | Use /CS1 |
| | 1x | Use /CS2 |

Table 3 describes the operation of the four memory bank control registers. The registers are write-only. Each register controls one quadrant in the 1M address space.

- Bits 7,6—The number of wait states used in access to this quadrant. Without wait states, read requires 2 clocks and write requires 3 clocks. The wait state adds to these numbers.

- Bits 5, 4—These bits allow the upper address lines to be inverted. This inversion occurs after the logic that selects the bank register, so setting these lines has no effect on which bank register is used. The inversion may be used to install a 1M memory chip in the space normally allocated to a 256K chip. The larger memory can then be accessed as 4 pages of 256K each. There is no effect outside the quadrant that the memory bank control register is controlling.

- Bit 3—Inhibits the write pulse to memory accessed in this quadrant. Useful for protecting flash memory from an inadvertent write pulse, which will not actually write to the flash because it is protected by lock codes, but will temporarily disable the flash memory and crash the system if the memory is used for code.

- Bit 2—Selects which set of the two lines /OEx and /WEx will be driven for memory accesses in this quadrant.

- Bits 1,0—Determines which of the three chip select lines will be driven for memory accesses to this quadrant.

- All bits of the control register are initialized to zero on reset.