

## TN241

## Accessing Large Memories and Bank-Switching with the Rabbit

This Technical Note describes memory bank-switching using the Rabbit microprocessor. Bank-switching is needed to map memory devices beyond the current 1MB limit\* imposed by the Dynamic C compiler and its libraries. The four sample programs provided in TN241.zip implement bank-switching. Each sample remaps the `xmem segment` (logical memory) to `Quadrant 3` (physical memory) to extend the range of addressable memory.

Each sample is hard-coded to use specific memory devices, but are general enough to modify to use another memory device. You would simply change the parameters that identify the memory control lines that are passed to the memory access functions. The memory control lines are the physical connections between the memory device and the Rabbit microprocessor: chip select (`/CSx`), output enable (`/OEx`) and write enable (`/WE`). If you are using a Rabbit module or single-board computer (SBC), the memory control lines are found on the schematic. For example, if you are using a PowerCore FLEX board, the memory control lines are connected as follows:

- Primary Flash: `/CS0, /OE0, /WE0`
- Secondary Flash: `/CS0, /OE1, /WE1`
- SRAM: `/CS1, /OE1, /WE1`
- Fast SRAM: `/CS2, /OE1, /WE1`

You cannot change the physical connection of these memory control lines unless you are doing board-level design. You can, however, use this information to identify which memory device is being accessed by the software.

### Accessing RAM: `anymem.c` and `writeunmappedram.c`

There are two sample programs that can be used to map up to 1MB of SRAM. Both should be compiled with separate I&D space disabled and “Code and BIOS in Flash” checked as the compile mode.

The sample `anymem.c` is written mostly in assembly language for speed. It accesses the primary flash chip so that it can run on all Rabbit-based targets, even those with 1 MB of memory or less. The program has a read function that can access any memory in the 6 MB space available to the Rabbit 2000 and 3000. It also has a write function that can access any RAM in the 6 MB space.

---

\* The 1MB limit does not include fast SRAM used for program execution (because it shares physical address space with the primary flash) or memory devices not connected to the Rabbit’s address lines, such as the serial Flash which uses SPI.

The second sample program, `writeunmappedram.c`, is written entirely in Dynamic C. It reads and writes the upper half of a large SRAM device using `root2xmem()` and `xmem2root()` functions calls.

**Accessing Flash:** `WriteUnmappedFlash.c` and `WriteUnmappedFlash2.c`

These programs map a secondary 512K Flash device (e.g., on a PowerCore FLEX board). The compile mode must be “Code and BIOS in Flash.”

**NOTE:** Be aware that Flash can only be written a finite number of times, depending on the Flash type. Wear leveling increases the life of the Flash device, but is not implemented in `WriteUnmappedFlash.c` or `WriteUnmappedFlash2.c`.

The sample `WriteUnmappedFlash.c` limits writes to sector size aligned physical addresses. That is, addresses must be a multiple of the sector size (e.g., an addresses for a 4K sector Flash must be evenly divisible by 4K). Furthermore, only “size” bytes will be written to the sector-aligned physical memory location.

The sample `WriteUnmappedFlash2.c` is not limited to sector-aligned writes. It can read and write arbitrary size buffers to arbitrary locations because it copies to RAM the Flash sector that the destination address lies within to the flash buffer, erases the entire sector, copies the data into the correct offset within the flash buffer, and then writes the entire sector back byte by byte. (Most Flash devices used by Rabbit are byte writable and sector erasable.)

## Memory Management Background

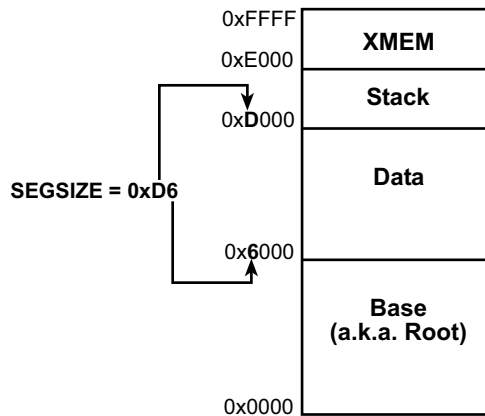
The rest of this Technical Note describes the basic memory management scheme used by the Rabbit. The information provided here will help you understand the internal workings of the sample programs listed above. For further details about Rabbit memory management, please refer to the [reference section](#) at the end of this document.

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

### Memory Mapping Unit (MMU)

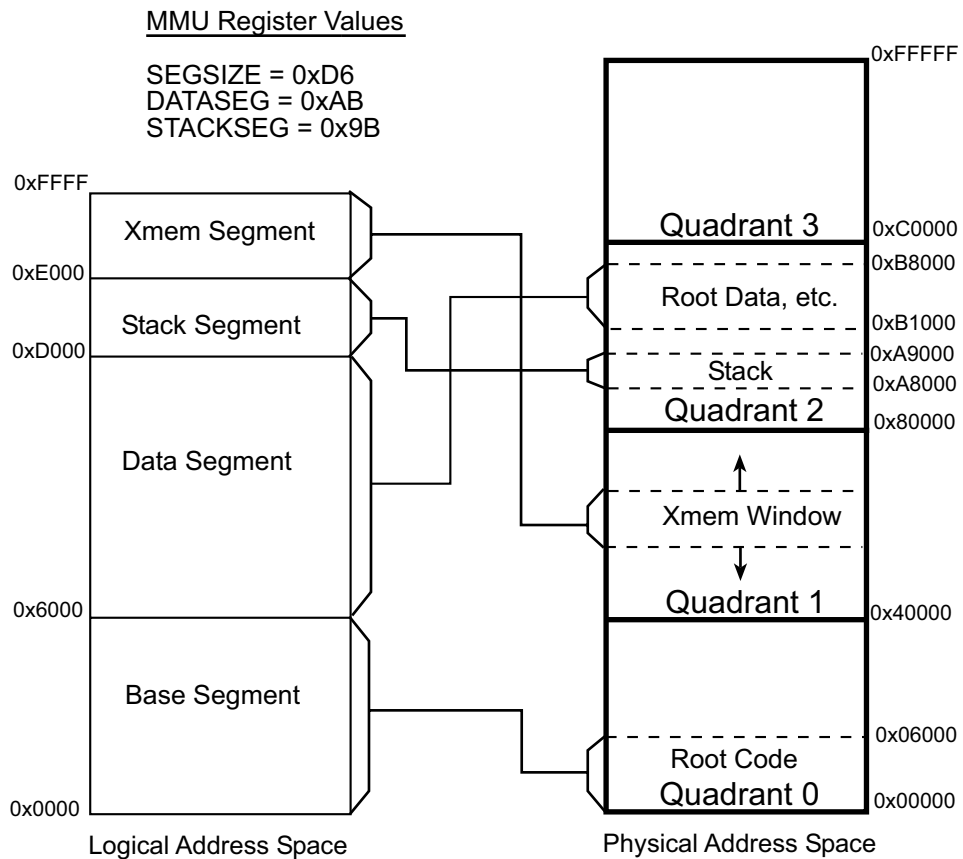
The MMU translates a 16-bit logical address to a 20-bit physical address. The interface to the MMU uses four registers, identified by the macros: `XPC`, `STACKSEG`, `DATASEG` and `SEGSIZE`.

**Figure 1. The Logical Address Space**



The logical address space is divided into four segments: xmem, stack, data and base. The xmem segment always occupies addresses 0xE000h to 0xFFFF, inclusive. The address range for the other three segments range from 0x0000 to 0xDFFF. The register SEGSIZE defines boundaries in the logical address space as shown in Figure 1. The other three MMU registers are the segment registers: XPC, STACKSEG and DATASEG. These registers are used to map logical addresses that fall within their associated segment to the physical address space. Figure 2 shows an example memory mapping. An actual memory mapping depends on board type and compilation options.

**Figure 2. Logical Address to Physical Address Mapping**



The physical address space is divided into quadrants (four banks); each one is 256K in size. Addresses within the base segment map directly to addresses in the physical address space starting from 0. For the other segments, the physical address is computed by shifting the segment register left 12 bits (multiplying by 0x1000) and adding the product to the 16-bit logical address. Given the register values for DATASEG and STACKSEG shown in [Figure 2](#), the equations are:

$$(0xAB * 0x1000) + 0x6000 = 0xB1000$$

and

$$(0x9B * 0x1000) + 0xD000 = 0xA8000$$

Therefore, 0xB1000 is the first address in the physical address space for the data segment and 0xA8000 is the first address in the physical address space for the stack segment.

The xmem segment was designed to be remapped at runtime and is usually the best choice when an application needs to access bank-switched memory.

Here is the algorithm the Rabbit processor uses to convert logical addresses to physical addresses.

LA = Logical Address: address within the 64K logical address space

PA = Physical Address: address within the 1MB physical address space

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

If LA >= E000h

$$PA = LA + (XPC * 1000h)$$

Else If LA >= X000h

$$PA = LA + (STACKSEG * 1000h)$$

Else If LA >= Y000h

$$PA = LA + (DATASEG * 1000h)$$

Else PA = LA

This algorithm is used to access the 1MB physical memory space. The memory bank control registers are manipulated to access more than 1MB.

## Memory Interface Unit (MIU)

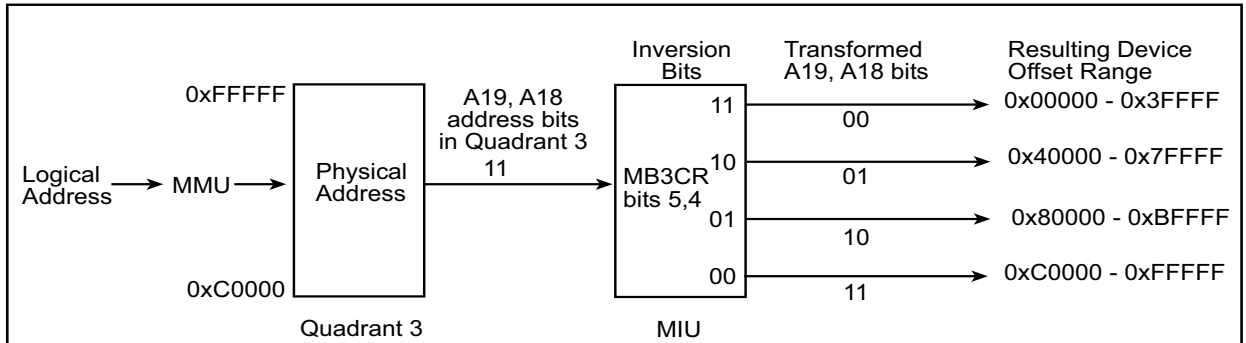
The MIU controls memory access after the MMU determines the physical address. There are five registers associated with the MIU: the MMIDR (MMU Instruction/Data Register) and the four memory bank control registers. Along with some timing options, the MMIDR controls two important functions: separate instruction and data space access and the permanent enabling of /CS1. For more information on these topics, see the *Rabbit 3000 Microprocessor User's Manual*.

Each of the four memory bank control registers (MB0CR, MB1CR, MB2CR and MB3CR) control one 256K quadrant of the 1 MB physical address space. These registers control wait states, memory control line usage and write protection. They also control inversion of address lines A18 and A19; the inversion is applied after the physical address is determined by the MMU.

[Figure 3](#) illustrates how address line inversion is used to map a 256K quadrant to 1MB of memory. Each quadrant has a consistent bit pattern for the upper two address bits. The values of A19 and A18 default to “00” in Quadrant 0, and “01” in Quadrant 1, “10” in Quadrant 2 and “11” in Quadrant 3, respectively. By inverting these address lines after the quadrant has been determined, we can access a full megabyte of a memory device using only one quadrant.

You must select a quadrant to remap before you can calculate the inversions that correspond to the desired device offset. For example, in [Figure 3](#) every physical address between 0xC0000 and 0xFFFFF coming from the MMU falls within Quadrant 3, thus the inversions and control settings from MB3CR are applied.

**Figure 3. Using Address Line Inversion to Map 1MB of Physical Memory**



For example, if the device offset is 0x00000 and we want to use Quadrant 3 to access it, then A18 and A19 both need to be inverted. The inversion of A18 and A19 results in the physical address 0xC0000 mapping to the device offset 0x00000. By setting the inversion bits (bits 4 and 5) of the MB3CR, the quadrant is mapped to the bottom 256K of the memory device. In other words, the physical address range 0xC0000 through 0xFFFFF maps to 0x00000 through 0x3FFFF of the memory device. To map the next 256K of the memory device, we need to pass A18 unchanged and invert A19. Therefore, bit 4 of MB3CR must be zero and bit 5 is one. Doing the math again, we see that 0xC0000 through 0xFFFFF is mapped to 0x40000 through 0x7FFFF.

By knowing the address on the device (i.e., the offset from the beginning of the device) and the quadrant, we can calculate the bit values that must be placed in the corresponding MBxCR to ensure that the correct values are placed on the address lines A18 and A19 when the device is being accessed. The following equation can be used for this calculation:

$$\text{MBXCR inversion bits [5:4]} = (\text{0xNNNNN} / \text{0x40000}) \text{ XOR QUAD}$$

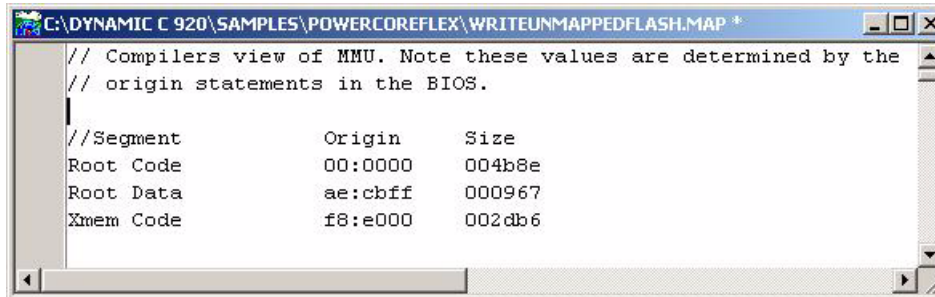
where 0xNNNNN is any device offset in a 1MB space and 0x40000 is the quadrant size. The result of the division is then XOR'd with QUAD, which is 0x0, 0x1, 0x2 or 0x3 to specify Quadrant 0, 1, 2 or 3. For example, if we want to write to device offset 0x55000 through quadrant 2 then the inversion bits would be  $(0x55000/0x40000) \text{ XOR } 2 = 0x3$ . So bits 5 and 4 of MB2CR would both be set high.

The sample programs included in TN241.zip use Quadrant 3 because it is generally a safe quadrant to remap. You can determine which quadrants are safe, i.e., available for remapping, by looking at the map file that is generated when you compile an application. Using the information found in the map file and the values of the segment registers, you can determine where the logical address space segments are mapped for code and data. (HINT: The sample programs turn interrupts off to prevent any interrupt dependent code and data potentially located in the bank-switched-out quadrant from crashing the program. Alternatively, interrupts can be enabled if code analysis can show that the bank-switched-out quadrant contains no interrupt dependent code or data.)

## Understanding the Logical to Physical Mapping

In Figure 4 is an example of the information reported at the top of a map file. The starting address and size of the base segment (“Root Code”) are shown first. Next is the last address in the data segment (“Root Data”) and its size. The last address is shown because the data segment will grow down in physical memory space. Next is the first address of the xmem segment (“Xmem Code”) and its size.

Figure 4. Memory Mapping Information



```
C:\DYNAMIC C 920\SAMPLES\POWERCOREFLEX\WRITEUNMAPPEDFLASH.MAP *
// Compilers view of MMU. Note these values are determined by the
// origin statements in the BIOS.
//Segment          Origin      Size
Root Code          00:0000    004b8e
Root Data          ae:cbff    000967
Xmem Code          f8:e000    002db6
```

From looking at the map file, you know the values of DATASEG and XPC, which are 0xAE and 0xF8, respectively, in the above screenshot. The physical address of the stack segment is not shown in the map file, but must be determined by reading the values of STACKSEG and SEGSIZE at runtime. Using the “Evaluate Expression” feature of the debugger you can easily check these register values and use this information to figure out which quadrants the data, stack and xmem segments are mapped to. E.g., type

```
RdPortI (SEGSIZE)
```

into the evaluate expression box to see the value of SEGSIZE.

Typically, the XPC register changes frequently during runtime to access the xmem code and data regions shown in the map file as well as data allocated by `xalloc()` and `xavail()` (for RAM only). Typically, DATASEG and STACKSEG\*\* do not change during runtime.

To map addresses from the xmem segment into a re-mapped quadrant, we need to set the XPC register. For the purpose of discussion we will use Quadrant 3 assuming we have setup MB3CR with the appropriate value to point to the desired 256k bank of the memory device of choice.

Next, we compute the physical address using the physical offset in the device. As an example of the general case, we will use 0x81500 as the device offset. We perform the calculation by masking the off the upper two bits of the device offset and adding the originating address of the remapped quadrant.

$$(0x81500 \& 0x3FFFF) + 0xC0000 = 0xC1500$$

We know from the MMU that if the logical address falls in the xmem segment ( $LA \geq 0xE000$ ), the physical address is calculated using the following equation:

$$PA = LA + (XPC * 0x1000)$$

Therefore, to map a physical address into the xmem window we simply subtract 0xE000 from the address and shift the result by 12 bits to the right to get the 8 bit XPC.

$$(0xC1500 - 0xE000) \gg 12 = 0xB3$$

---

\*\* STACKSEG will change during runtime when using  $\mu$ C/OS-II; therefore, precautions must be taken to ensure the stack is safe.

If you examine the sample programs provided in TN241.zip, you will see the above calculation for the XPC register in the code.

## Conclusion

The sample programs provided in TN241.zip allow access to memory devices beyond the 1MB physical memory space mapped by Dynamic C. Applications such as data loggers that need to access extra RAM can make use of the code in `anymem.c` or `WriteUnmappedRAM.c`. The two programs that access unmapped Flash can be used as a template for an application that needs to store images or large numbers of constants. They can also be used in producing something more complex, such as a download manager to write a program to the second Flash device.

## References

There are several additional documents that contain information about Rabbit memory management.

- *Rabbit 3000 Microprocessor User's Manual*
- *Rabbit 3000 Designer's Handbook*
- Tech Note: TN202, "Rabbit Memory Management"
- Tech Note: TN219, "Root Memory Usage Reduction Tips"
- Tech Note: TN238, "Rabbit Memory Usage Tips"

These documents are available on the CD containing Dynamic C and on the Rabbit Semiconductor website. The Rabbit 3000 manuals can be found here:

[www.rabbitsemiconductor.com/docs/](http://www.rabbitsemiconductor.com/docs/)

The Technical Notes can be found here:

[www.rabbitsemiconductor.com/support/techNotes\\_whitePapers.shtml](http://www.rabbitsemiconductor.com/support/techNotes_whitePapers.shtml)