Generating ARM MMU Page Tables with MMUgen

Mmugen is a utility program that creates MMU tables for ARM's cached processors. The program reads a text file describing the target system's memory map and generates a binary file containing level-one and level-two MMU page tables. The "INCBIN" assembler directive may then be used to include the binary file into the final executable file.

MMUgen could also be adapted to generate MMU tables on a target system at run time. This reduces the ROM requirements dramatically, since rule files are much smaller than the generated MMU tables, and can be compressed still further by encoding the rules as C-structures (MMUgen does this encoding as an interim step in its current form).

As supplied, MMUgen may be compiled as a native application on any system with an ANSI standard compiler, or may be run on ARMulator, using Semihosting to read data from the rules text file and to write back the binary MMU tables on the Host system. MMUgen is a command-line utility. The command line format is: `mmugen rulesfile outputfile`

MMUgen features

MMUgen sets all level-one entries to "domain 0" and assumes that domain zero is set up for client-permissions. MMUgen could be readily extended to allow domains to be specified for each memory region.

The MMUgen rules file allows the user to specify for each region of the virtual address space:

- The corresponding physical address for the region
- Whether the region maps on to memory on the target board, or if the MMU should generate an exception if the region is accessed
- The access permissions for the region:
  - FULL_ACCESS (any process can read or write in that region)
  - SVC_READWRITE (privileged modes have full access, user mode has no access)
  - NO_USR_WRITE (privileged modes have full access, user mode has only read access)
  - NO_ACCESS (neither privileged or user modes have read or write access to the region)
- Whether the region is cacheable
- Whether the Write Buffer is enabled for the region.

MMUgen Rules File Format

C-style comments (using /* … */) are permitted anywhere in an MMUgen rules file.

The general format of a rules files is:

```
BASE_ADDRESS    ……
LEVEL 1
        …..
LEVEL 2
        …..
POSTPROCESS     ……
```

BASE_ADDRESS

The BASE_ADDRESS entry must be specified; this is the address at which your MMU table will be placed in the physical memory map. The address must be on a 16K byte boundary. The base address is used when generating level-1 table entries that refer to level-2 page tables.

All addresses in the rules file are expected to be in hexadecimal format.

LEVEL 1

In the LEVEL 1 section, the complete 4-gigabyte address space is specified. MMUgen uses the descriptions in this section to generate 4096 table entries, each of which describes one megabyte in the virtual address space. The virtual and physical addresses in this section must therefore be on one megabyte boundaries. Typical entries are:

```
/* 1MB FLASH mapped to RAM */ VIRTUAL 0x00000000 TO 0x000FFFFF PHYSICAL 0xC0400000  PAGES
:
/* External hardware   */ VIRTUAL 0x50200000 TO 0x503FFFFF PHYSICAL 0x50200000  SECTION
FULL_ACCESS  NOT CACHEABLE AND NOT BUFFERABLE
:
/* Gap to 0x80000000   */ VIRTUAL 0x60100000 TO 0x7FFFFFFF PHYSICAL 0x60100000  FAULT
```

There would be other entries between the ones shown here; it is important that the entries in the Level 1 section are in virtual-address order, and that no gaps are left in the description of the memory map, otherwise an incorrect table will generated.

Note that regions marked with the PAGES keyword should be exactly one megabyte long, but SECTION and FAULT regions can describe larger regions.

The first entry describes a megabyte of virtual memory space that will be mapped onto RAM starting at 0xC0400000. In the system that this example is taken from, the RAM is not contiguous, so we have to specify further information (in the Level 2 section) that allows us to close up the gaps in the physical memory space. We tell MMUgen we want to specify further information by using the word “PAGES”. This may also be required if we want to set different access permissions for regions of memory smaller than 1 megabyte, or if the physical memory present is smaller than a 1 megabyte chunk.

The second entry corresponds to a 2-megabyte region of hardware expansion space. This is one-to-one mapped, has full access permissions for both privileged and user mode applications, and is not cached or write-buffered.

The third entry describes a region of virtual memory for which no corresponding physical memory exists. The physical address specified is not important; the FAULT keyword will generate table entries that will cause the MMU to signal an abort exception to the ARM processor for any access to this virtual memory region.

LEVEL 2

This section is required only if any “PAGES” regions have been specified in the LEVEL 1 section. Remember that each “PAGES” region in the LEVEL 1 section must describe exactly one megabyte of the virtual address space. The corresponding entries in the LEVEL 2 section also describe exactly one megabyte, and are defined ***in the same order that they appear in the LEVEL 1 section***.

These rules allow MMUgen to calculate where the corresponding level 2 page tables will appear while generating the level 1 page table entries.

It's worth noting that the level 1 table generated by MMUgen will occupy 16K bytes of memory, and each PAGES reference in the rules table LEVEL 1 section will cause a further level 2 page table to be generated, occupying another 1K bytes of memory.

Typical Level 2 rules file entries would be:

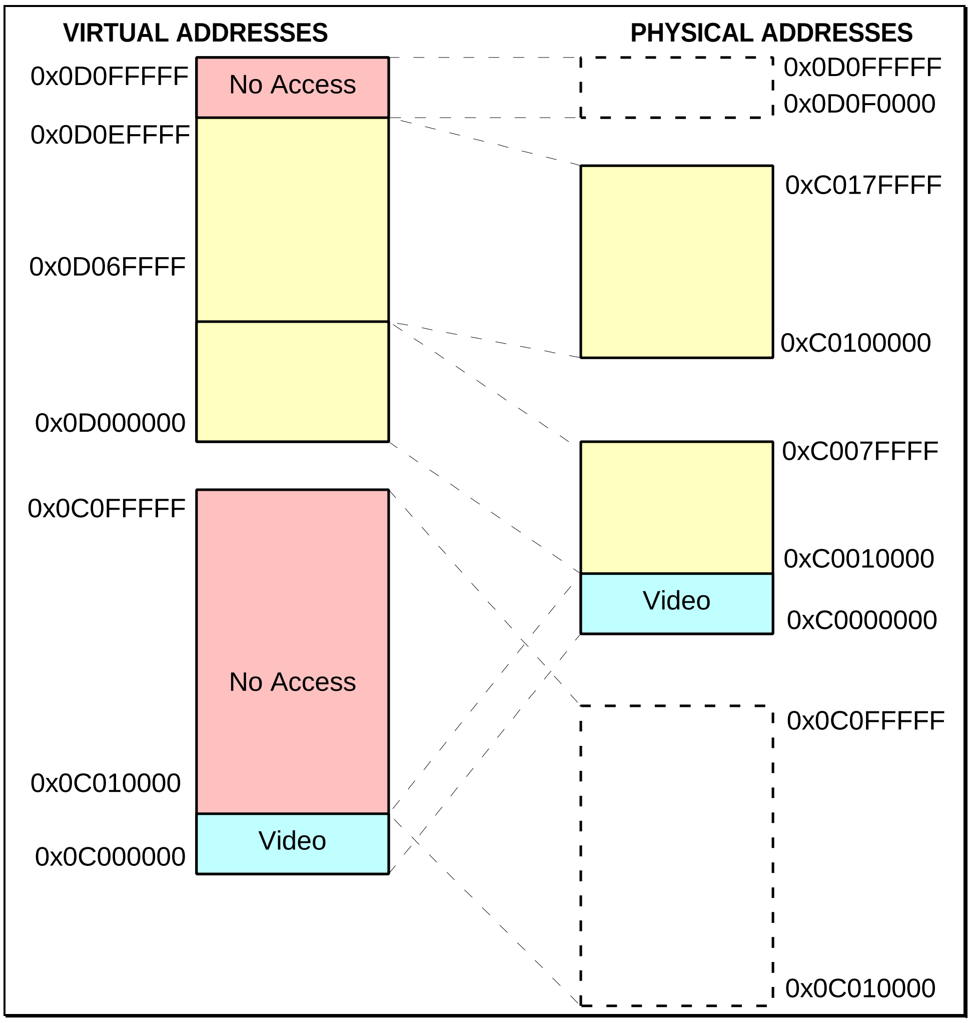
```
/* FLASH shadow image  */ VIRTUAL 0x00000000 TO 0x0007FFFF PHYSICAL 0xC0400000  LARGEPAGES
FULL_ACCESS      CACHEABLE AND BUFFERABLE
/* FLASH shadow image  */ VIRTUAL 0x00080000 TO 0x000FFFFF PHYSICAL 0xC0500000  LARGEPAGES
FULL_ACCESS      CACHEABLE AND BUFFERABLE

/* Video RAM RAM bank0  */ VIRTUAL 0x0C000000 TO 0x0C00FFFF PHYSICAL 0xC0000000  LARGEPAGES
FULL_ACCESS      CACHEABLE AND BUFFERABLE
                    VIRTUAL 0x0C010000 TO 0x0C0FFFFF PHYSICAL 0x0C010000  LARGEPAGES  NO_ACCESS
       NOT CACHEABLE AND NOT BUFFERABLE

/* RAM disk - RAM bank0 */ VIRTUAL 0x0D000000 TO 0x0D06FFFF PHYSICAL 0xC0010000  LARGEPAGES
FULL_ACCESS      CACHEABLE AND BUFFERABLE
/* RAM disk - RAM bank1*/ VIRTUAL 0x0D070000 TO 0x0D0EFFFF PHYSICAL 0xC0100000  LARGEPAGES
FULL_ACCESS      CACHEABLE AND BUFFERABLE
/* RAM disk - no RAM   */ VIRTUAL 0x0D0F0000 TO 0x0D0FFFFF PHYSICAL 0x0D0F0000  LARGEPAGES
NO_ACCESS    NOT CACHEABLE AND NOT BUFFERABLE
```

The first two entries describe the page entries corresponding to the first megabyte of virtual memory. The virtual addresses are not used by MMUgen, except to tell it how many pagetable entries to write for each descriptor. In this system, RAM appears in the physical memory map in 512K chunks, with gaps between each chunk, therefore two descriptors are required to give one megabyte of contiguous RAM in the virtual address space.

The descriptors for “Video RAM” and “RAM disk” describe 2-megabytes of virtual address space, where only one megabyte of memory is physically present in the memory map.



In the examples shown above, LARGEPAGES specifies that MMUgen should generate level 2 table entries for 64K pages. If we need to define a memory map with finer granularity, we can specify SMALLPAGES. This allows us to specify regions with 4K pages. This does not affect the size of the page tables generated by MMUgen.

ARM Architecture 5T also has the capability to use “tiny” pages – each table entry specifies 1K in the memory map. MMUgen does not yet provide support for tiny pages.

POSTPROCESS

The (optional) final section of an MMUgen rules file allows the MMU table to be automatically modified after it has been generated by the descriptors in the Level 1 and Level 2 sections.

Although POSTPROCESS would never normally be used, expanding on the examples shown above will show how POSTPROCESS can be useful in a real system.

In our example system, we want to copy code out of relatively slow FLASH, and execute it from RAM. We therefore need to map some memory starting at virtual address 0x00000000 onto the physical address of our RAM. As we’ve seen, the RAM is inconveniently segmented, so we need to use level-2 pagetables to make the RAM appear to be contiguous in the virtual memory map. So, in the level-1 section of the rules file, the first entry is marked with “LARGEPAGES”, and descriptors are included in the level-2 section to complete the virtual to physical mapping.

On reset however, we want to execute from FLASH (at physical address 0x00000000), until the program image has been copied to RAM. This would not normally be a problem – we can copy the code over before enabling the MMU. In our example system however, the external DRAM is not available while the MMU and writebuffer is disabled. This may seem a contrived example, but has been seen in a real system.

So we have two options:
- either to have two complete MMU tables in FLASH - differing only in the first table entry,
- or to set up the table for the correct operation before copying code to RAM and subsequently modify the first entry of the MMU table when the code has been copied.

Remember that Level 2 page tables must be defined in the order that the “PAGES” descriptors are listed in the Level 1 section of the rules file. Therefore, in this example we must set the first descriptor to generate an MMU table in the form in which it will be required when running from RAM.

The following POSTPROCESS directive can be used to modify the first entry of the MMU table, changing it to full-access, cacheable and bufferable, mapping the first megabyte of the virtual address space onto the first megabyte of physical address space:

```
POSTPROCESS ENTRY 0x00000000 EQUALS 0x00000C1A
```

This literally stores the value 0x00000C1A at address 0x00000000 bytes into the MMU table.

Example Files

EXAMPLE1.RULES

This file can be used to generate an extremely simple memory map – 4-gigabytes of virtual address space with a one-to-one mapping onto physical memory. All memory is fully accessible, and cached and write-buffered.

The file contains the BASE_ADDRESS directive, and just one descriptor line in the Level 1 section, describing the full 4-gigabyte memory space.

EXAMPLE2.RULES

This is quite a complex example based on a real system. The example descriptors shown in the main text of this document were taken from this memory map.

The system had 2MB of FLASH, 1MB for code and 1MB used as a FLASH disk drive.

4MB of RAM was fitted, of which 1MB was reserved for a shadow-copy of the FLASH code, 1MB was shared by the video buffer and a RAM disk, and 2MB was available for stacks, heap and C variables.

The RAM was not contiguous in physical memory, but instead appears as eight half-megabyte chunks spread over more than 22MB of address space. Level 2 page tables are needed to make the RAM appear contiguous in the virtual address space.

The virtual to physical address mapping carried out by the MMU means that the chunks of RAM can be used in any order. There is another hardware restriction, in that the video frame buffer must be at 0xC0000000 in the physical memory map. The video frame buffer in this system is 19.2K bytes long, but small pages are used to reserve 32K bytes, allowing for future expansion.

Small Pages are also used when setting access permissions for the hardware registers, which are grouped into 2K byte blocks. All other Level 2 page tables are set as Large Pages.

EXAMPLE3.RULES

This is for the same hardware as EXAMPLE2.RULES, but allows a more efficient use of RAM to be made. Instead of reserving a complete 1-megabyte block of RAM for the shadow copy of the FLASH code, three megabytes of RAM are mapped down to 0x00000000, and the "RW-BASE" is removed from the ARM linker options. This allows the linker to assume that the read-write data can follow on immediately from the code image, which is indeed the case once the FLASH image has been copied to RAM for execution.

The area reserved for the video frame buffer is also extended, allowing the privileged mode stacks to be placed above the frame buffer. This area was effectively unused in Example2.

MMUgen Error Messages

Usage: MMUgen Rulefile Outputfile	An incorrect number of command line arguments were specified; ensure that both the rules file and an output file have been specified.
Couldn't open Rule file	Ensure the correct path to the rules file was specified – when running from AXD, you need to specify “..\..\Example1.rules” if the rules file is in the same directory as the project and source files.
Couldn't open Output file	Ensure disk isn’t full or write-protected etc.
Table data error - at end of Level 1, the table was xxxxxxxx bytes long (should be 0x4000)	MMUgen performs a check to ensure that 4096 table entries have been generated from the Level 1 descriptors. The number of bytes shown in the error message will indicate if there were too many or too few entries generated. Check that the rules in your rules file span the complete 4-gigabyte virtual address range, without duplicating specification of any particular region.
Unrecognised word	MMUgen has found a word in the rules file that was not in its keyword list.
MMU page table level not set	The “LEVEL” keyword was not found before the first memory map descriptor was read from the rules file.
MMU Table synchronisation error! table_entry = xxxxxxxx, v_base = yyyyyyyy	This error will be reported if a virtual address specified in the Level 1 section of the rules file does not correspond to the virtual address mapping onto the page table entry currently being generated. Look for any gaps or duplications in the virtual addresses specified in your rules file.