FR451 MMU LINUX MEMORY MANAGEMENT

MMU HARDWARE

FR451 MMU Linux puts the MMU into EDAT mode whilst running. This means that it uses both the SAT
registers and the DAT TLB to perform address translation.

There are 8 IAMLR/IAMPR register pairs and 16 DAMLR/DAMPR register pairs for SAT mode.

In DAT mode, there is also a TLB organised in cache format as 64 lines x 2 ways. Each line spans a
16KB range of addresses, but can match a larger region.

MEMORY MANAGEMENT REGISTERS

Certain control registers are used by the kernel memory management routines:

REGISTERS	USAGE
IAMR0, DAMR0	Kernel image and data mappings
IAMR1, DAMR1	First-chance TLB lookup mapping
DAMR2	Page attachment for cache flush by page
DAMR3	Current PGD mapping
SCR0, DAMR4	Instruction TLB PGE/PTD cache
SCR1, DAMR5	Data TLB PGE/PTD cache
DAMR6-10	kmap_atomic() mappings
DAMR11	I/O mapping
CXNR	mm_struct context ID
TTBR	Page directory (PGD) pointer (physical address)

GENERAL MEMORY LAYOUT

The physical memory layout is as follows:

PHYSICAL ADDRESS	CONTROLLER	DEVICE
00000000 - BFFFFFFF	SDRAM	SDRAM area
E0000000 - EFFFFFFF	L-BUS CS2#	VDK SLBUS/PCI window
F0000000 - F0FFFFFF	L-BUS CS5#	MB93493 CSC area (DAV daughter board)
F1000000 - F1FFFFFF	L-BUS CS7#	(CB70 CPU-card PCMCIA port I/O space)
FC000000 - FC0FFFFF	L-BUS CS1#	VDK MB86943 config space
FC100000 - FC1FFFFF	L-BUS CS6#	DM9000 NIC I/O space
FC200000 - FC2FFFFF	L-BUS CS3#	MB93493 CSR area (DAV daughter board)

```
  FD000000 - FDFFFFFF   L-BUS CS4#      (CB70 CPU-card extra flash space)
  FE000000 - FEFFFFFF                   Internal CPU peripherals
  FF000000 - FF1FFFFF   L-BUS CS0#      Flash 1
  FF200000 - FF3FFFFF   L-BUS CS0#      Flash 2
  FFC00000 - FFC0001F   L-BUS CS0#      FPGA
```

The virtual memory layout is:

```
  VIRTUAL ADDRESS    PHYSICAL   TRANSLATOR      FLAGS   SIZE    OCCUPATION
  =================  ========   ==============  ======= ======= 
===================================
  00004000-BFFFFFFF  various    TLB,xAMR1       D-N-??V 3GB     Userspace
  C0000000-CFFFFFFF  00000000   xAMPR0          -L-S--V 256MB   Kernel image and 
data
  D0000000-D7FFFFFF  various    TLB,xAMR1       D-NS??V 128MB   vmalloc area
  D8000000-DBFFFFFF  various    TLB,xAMR1       D-NS??V 64MB    kmap() area
  DC000000-DCFFFFFF  various    TLB                     1MB     Secondary 
kmap_atomic() frame
  DD000000-DD27FFFF  various    DAMR                    160KB   Primary 
kmap_atomic() frame
  DD040000                      DAMR2/IAMR2     -L-S--V page    Page cache flush 
attachment point
  DD080000                      DAMR3           -L-SC-V page    Page Directory 
(PGD)
  DD0C0000                      DAMR4           -L-SC-V page    Cached insn TLB 
Page Table lookup
  DD100000                      DAMR5           -L-SC-V page    Cached data TLB 
Page Table lookup
  DD140000                      DAMR6           -L-S--V page    
kmap_atomic(KM_BOUNCE_READ)
  DD180000                      DAMR7           -L-S--V page    
kmap_atomic(KM_SKB_SUNRPC_DATA)
  DD1C0000                      DAMR8           -L-S--V page    
kmap_atomic(KM_SKB_DATA_SOFTIRQ)
  DD200000                      DAMR9           -L-S--V page    
kmap_atomic(KM_USER0)
  DD240000                      DAMR10          -L-S--V page    
kmap_atomic(KM_USER1)
  E0000000-FFFFFFFF  E0000000   DAMR11          -L-SC-V 512MB   I/O region
```

IAMPR1 and DAMPR1 are used as an extension to the TLB.

```
====================
KMAP AND KMAP_ATOMIC
====================
```

To access pages in the page cache (which may not be directly accessible if
highmem is available),
the kernel calls kmap(), does the access and then calls kunmap(); or it calls
kmap_atomic(), does
the access and then calls kunmap_atomic().

kmap() creates an attachment between an arbitrary inaccessible page and a range
of virtual
addresses by installing a PTE in a special page table. The kernel can then

access this page as it
wills. When it's finished, the kernel calls kunmap() to clear the PTE.

kmap_atomic() does something slightly different. In the interests of speed, it chooses one of two
strategies:

 (1) If possible, kmap_atomic() attaches the requested page to one of DAMPR5 through DAMPR10
     register pairs; and the matching kunmap_atomic() clears the DAMPR. This makes high memory
     support really fast as there's no need to flush the TLB or modify the page tables. The DAMLR
     registers being used for this are preset during boot and don't change over the lifetime of the
     process. There's a direct mapping between the first few kmap_atomic() types, DAMR number and
     virtual address slot.

     However, there are more kmap_atomic() types defined than there are DAMR registers available,
     so we fall back to:

 (2) kmap_atomic() uses a slot in the secondary frame (determined by the type parameter), and then
     locks an entry in the TLB to translate that slot to the specified page. The number of slots is
     obviously limited, and their positions are controlled such that each slot is matched by a
     different line in the TLB. kunmap() ejects the entry from the TLB.

Note that the first three kmap atomic types are really just declared as placeholders. The DAMPR
registers involved are actually modified directly.

Also note that kmap() itself may sleep, kmap_atomic() may never sleep and both always succeed;
furthermore, a driver using kmap() may sleep before calling kunmap(), but may not sleep before
calling kunmap_atomic() if it had previously called kmap_atomic().

USING MORE THAN 256MB OF MEMORY

The kernel cannot access more than 256MB of memory directly. The physical layout, however, permits
up to 3GB of SDRAM (possibly 3.25GB) to be made available. By using CONFIG_HIGHMEM, the kernel can
allow userspace (by way of page tables) and itself (by way of kmap) to deal with the memory
allocation.

External devices can, of course, still DMA to and from all of the SDRAM, even if the kernel can't

see it directly. The kernel translates page references into real addresses for communicating to the
devices.

```
===================
PAGE TABLE TOPOLOGY
===================
```

The page tables are arranged in 2-layer format. There is a middle layer (PMD) that would be used in
3-layer format tables but that is folded into the top layer (PGD) and so consumes no extra memory
or processing power.

```
  +-------+     PGD     PMD
  | TTBR |--->+-------------------+
  +-------+   |      |       : STE |
              | PGE0 | PME0  : STE |
              |      |       : STE |
              +-------------------+              Page Table
              |      |       : STE -------------->+---------+ +0x0000
              | PGE1 | PME0  : STE -----------+    | PTE0    |
              |      |       : STE -------+    |    +---------+
              +-------------------+        |    |    | PTE63   |
              |      |       : STE |       |    +-->+---------+ +0x0100
              | PGE2 | PME0  : STE |       |         | PTE64   |
              |      |       : STE |       |         +---------+
              +-------------------+        |         | PTE127  |
              |      |       : STE |       +------->+---------+ +0x0200
              | PGE3 | PME0  : STE |                 | PTE128  |
              |      |       : STE |                 +---------+
              +-------------------+                  | PTE191  |
                                                     +---------+ +0x0300
```

Each Page Directory (PGD) is 16KB (page size) in size and is divided into 64 entries (PGEs). Each
PGE contains one Page Mid Directory (PMD).

Each PMD is 256 bytes in size and contains a single entry (PME). Each PME holds 64 FR451 MMU
segment table entries of 4 bytes apiece. Each PME "points to" a page table. In practice, each STE
points to a subset of the page table, the first to PT+0x0000, the second to PT+0x0100, the third to
PT+0x200, and so on.

Each PGE and PME covers 64MB of the total virtual address space.

Each Page Table (PTD) is 16KB (page size) in size, and is divided into 4096 entries (PTEs). Each
entry can point to one 16KB page. In practice, each Linux page table is subdivided into 64 FR451
MMU page tables. But they are all grouped together to make management easier, in particular rmap
support is then trivial.

Grouping page tables in this fashion makes PGE caching in SCR0/SCR1 more efficient because the
coverage of the cached item is greater.

Page tables for the vmalloc area are allocated at boot time and shared between all mm_structs.

=================
USER SPACE LAYOUT
=================

For MMU capable Linux, the regions userspace code are allowed to access are kept entirely separate
from those dedicated to the kernel:

```
        VIRTUAL ADDRESS    SIZE   PURPOSE
        =================  =====  ===================================
        00000000-00003fff  4KB    NULL pointer access trap
        00004000-01ffffff  ~32MB  lower mmap space (grows up)
        02000000-021fffff  2MB    Stack space (grows down from top)
        02200000-nnnnnnnn         Executable mapping
        nnnnnnnn-                 brk space (grows up)
                -bfffffff         upper mmap space (grows down)
```

This is so arranged so as to make best use of the 16KB page tables and the way in which PGEs/PMEs
are cached by the TLB handler. The lower mmap space is filled first, and then the upper mmap space
is filled.

===============================
GDB-STUB MMU DEBUGGING SERVICES
===============================

The gdb-stub included in this kernel provides a number of services to aid in the debugging of MMU
related kernel services:

 (*) Every time the kernel stops, certain state information is dumped into __debug_mmu. This
     variable is defined in arch/frv/kernel/gdb-stub.c. Note that the gdbinit file in this
     directory has some useful macros for dealing with this.

     (*) __debug_mmu.tlb[]

         This receives the current TLB contents. This can be viewed with the _tlb GDB macro:

```
                (gdb) _tlb
                tlb[0x00]: 01000005 00718203  01000002 00718203
                tlb[0x01]: 01004002 006d4201  01004005 006d4203
                tlb[0x02]: 01008002 006d0201  01008006 00004200
                tlb[0x03]: 0100c006 007f4202  0100c002 0064c202
                tlb[0x04]: 01110005 00774201  01110002 00774201
                tlb[0x05]: 01114005 00770201  01114002 00770201
                tlb[0x06]: 01118002 0076c201  01118005 0076c201
                ...
                tlb[0x3d]: 010f4002 00790200  001f4002 0054ca02
                tlb[0x3e]: 010f8005 0078c201  010f8002 0078c201
                tlb[0x3f]: 001fc002 0056ca01  001fc005 00538a01
```

 (*) __debug_mmu.iamr[]
 (*) __debug_mmu.damr[]

These receive the current IAMR and DAMR contents. These can be viewed with the _amr
GDB macro:

```
                (gdb) _amr
                AMRx           DAMR                       IAMR
                ====   =====================   =====================
                amr0 : L:c0000000 P:00000cb9 : L:c0000000 P:000004b9
                amr1 : L:01070005 P:006f9203 : L:0102c005 P:006a1201
                amr2 : L:d8d00000 P:00000000 : L:d8d00000 P:00000000
                amr3 : L:d8d04000 P:00534c0d : L:00000000 P:00000000
                amr4 : L:d8d08000 P:00554c0d : L:00000000 P:00000000
                amr5 : L:d8d0c000 P:00554c0d : L:00000000 P:00000000
                amr6 : L:d8d10000 P:00000000 : L:00000000 P:00000000
                amr7 : L:d8d14000 P:00000000 : L:00000000 P:00000000
                amr8 : L:d8d18000 P:00000000
                amr9 : L:d8d1c000 P:00000000
                amr10: L:d8d20000 P:00000000
                amr11: L:e0000000 P:e0000ccd
```

 (*) The current task's page directory is bound to DAMR3.

This can be viewed with the _pgd GDB macro:

```
           (gdb) _pgd
           $3 = {{pge = {{ste = {0x554001, 0x554101, 0x554201, 0x554301, 0x554401,
                     0x554501, 0x554601, 0x554701, 0x554801, 0x554901, 0x554a01,
                     0x554b01, 0x554c01, 0x554d01, 0x554e01, 0x554f01, 0x555001,
                     0x555101, 0x555201, 0x555301, 0x555401, 0x555501, 0x555601,
                     0x555701, 0x555801, 0x555901, 0x555a01, 0x555b01, 0x555c01,
                     0x555d01, 0x555e01, 0x555f01, 0x556001, 0x556101, 0x556201,
                     0x556301, 0x556401, 0x556501, 0x556601, 0x556701, 0x556801,
                     0x556901, 0x556a01, 0x556b01, 0x556c01, 0x556d01, 0x556e01,
                     0x556f01, 0x557001, 0x557101, 0x557201, 0x557301, 0x557401,
                     0x557501, 0x557601, 0x557701, 0x557801, 0x557901, 0x557a01,
                     0x557b01, 0x557c01, 0x557d01, 0x557e01, 0x557f01}}}}, {pge =
{{
                ste = {0x0 <repeats 64 times>}}}} <repeats 51 times>, {pge =
{{ste = {
                     0x248001, 0x248101, 0x248201, 0x248301, 0x248401, 0x248501,
                     0x248601, 0x248701, 0x248801, 0x248901, 0x248a01, 0x248b01,
                     0x248c01, 0x248d01, 0x248e01, 0x248f01, 0x249001, 0x249101,
                     0x249201, 0x249301, 0x249401, 0x249501, 0x249601, 0x249701,
                     0x249801, 0x249901, 0x249a01, 0x249b01, 0x249c01, 0x249d01,
                     0x249e01, 0x249f01, 0x24a001, 0x24a101, 0x24a201, 0x24a301,
                     0x24a401, 0x24a501, 0x24a601, 0x24a701, 0x24a801, 0x24a901,
                     0x24aa01, 0x24ab01, 0x24ac01, 0x24ad01, 0x24ae01, 0x24af01,
                     0x24b001, 0x24b101, 0x24b201, 0x24b301, 0x24b401, 0x24b501,
                     0x24b601, 0x24b701, 0x24b801, 0x24b901, 0x24ba01, 0x24bb01,
                     0x24bc01, 0x24bd01, 0x24be01, 0x24bf01}}}}, {pge = {{ste = {
                     0x0 <repeats 64 times>}}}} <repeats 11 times>}
```

 (*) The PTD last used by the instruction TLB miss handler is attached to DAMR4.
 (*) The PTD last used by the data TLB miss handler is attached to DAMR5.

These can be viewed with the _ptd_i and _ptd_d GDB macros:

```
        (gdb) _ptd_d
        $5 = {{pte = 0x0} <repeats 127 times>, {pte = 0x539b01}, {
            pte = 0x0} <repeats 896 times>, {pte = 0x719303}, {pte = 0x6d5303},
{
            pte = 0x0}, {pte = 0x0}, {pte = 0x0}, {pte = 0x0}, {pte = 0x0}, {
            pte = 0x0}, {pte = 0x0}, {pte = 0x0}, {pte = 0x0}, {pte = 0x6a1303},
{
            pte = 0x0} <repeats 12 times>, {pte = 0x709303}, {pte = 0x0}, {pte =
0x0},
          {pte = 0x6fd303}, {pte = 0x6f9303}, {pte = 0x6f5303}, {pte = 0x0}, {
            pte = 0x6ed303}, {pte = 0x531b01}, {pte = 0x50db01}, {
            pte = 0x0} <repeats 13 times>, {pte = 0x5303}, {pte = 0x7f5303}, {
            pte = 0x509b01}, {pte = 0x505b01}, {pte = 0x7c9303}, {pte =
0x7b9303}, {
            pte = 0x7b5303}, {pte = 0x7b1303}, {pte = 0x7ad303}, {pte = 0x0}, {
            pte = 0x0}, {pte = 0x7a1303}, {pte = 0x0}, {pte = 0x795303}, {pte =
0x0}, {
            pte = 0x78d303}, {pte = 0x0}, {pte = 0x0}, {pte = 0x0}, {pte = 0x0},
{
            pte = 0x0}, {pte = 0x775303}, {pte = 0x771303}, {pte = 0x76d303}, {
            pte = 0x0}, {pte = 0x765303}, {pte = 0x7c5303}, {pte = 0x501b01}, {
            pte = 0x4f1b01}, {pte = 0x4edb01}, {pte = 0x0}, {pte = 0x4f9b01}, {
            pte = 0x4fdb01}, {pte = 0x0} <repeats 2992 times>}
```