
INTERNAL KERNEL ABI FOR FR-V ARCH

The internal FRV kernel ABI is not quite the same as the userspace ABI. A number of the registers are used for special purposed, and the ABI is not consistent between modules vs core, and MMU vs no-MMU.

This partly stems from the fact that FRV CPUs do not have a separate supervisor stack pointer, and most of them do not have any scratch registers, thus requiring at least one general purpose register to be clobbered in such an event. Also, within the kernel core, it is possible to simply jump or call directly between functions using a relative offset. This cannot be extended to modules for the displacement is likely to be too far. Thus in modules the address of a function to call must be calculated in a register and then used, requiring two extra instructions.

This document has the following sections:

- (*) System call register ABI
- (*) CPU operating modes
- (*) Internal kernel-mode register ABI
- (*) Internal debug-mode register ABI
- (*) Virtual interrupt handling

SYSTEM CALL REGISTER ABI

When a system call is made, the following registers are effective:

REGISTERS	CALL	RETURN
GR7	System call number	Preserved
GR8	Syscall arg #1	Return value
GR9-GR13	Syscall arg #2-6	Preserved

CPU OPERATING MODES

The FR-V CPU has three basic operating modes. In order of increasing capability:

- (1) User mode.

Basic userspace running mode.

- (2) Kernel mode.

Normal kernel mode. There are many additional control registers available that may be accessed in this mode, in addition to all the stuff available to user mode. This has two submodes:

(a) Exceptions enabled (PSR.T == 1).

Exceptions will invoke the appropriate normal kernel mode handler. On entry to the handler, the PSR.T bit will be cleared.

(b) Exceptions disabled (PSR.T == 0).

No exceptions or interrupts may happen. Any mandatory exceptions will cause the CPU to halt unless the CPU is told to jump into debug mode instead.

(3) Debug mode.

No exceptions may happen in this mode. Memory protection and management exceptions will be flagged for later consideration, but the exception handler won't be invoked. Debugging traps such as hardware breakpoints and watchpoints will be ignored. This mode is entered only by debugging events obtained from the other two modes.

All kernel mode registers may be accessed, plus a few extra debugging specific registers.

```
=====
INTERNAL KERNEL-MODE REGISTER ABI
=====
```

There are a number of permanent register assignments that are set up by entry.S in the exception prologue. Note that there is a complete set of exception prologues for each of user->kernel transition and kernel->kernel transition. There are also user->debug and kernel->debug mode transition prologues.

REGISTER	FLAVOUR	USE
GR1		Supervisor stack pointer
GR15		Current thread info pointer
GR16		GP-Rel base register for small data
GR28		Current exception frame pointer (__frame)
GR29		Current task pointer (current)
GR30		Destroyed by kernel mode entry
GR31	NOMMU	Destroyed by debug mode entry
GR31	MMU	Destroyed by TLB miss kernel mode entry
CCR. ICC2		Virtual interrupt disablement tracking
CCCR. CC3		Cleared by exception prologue (atomic op emulation)
SCR0	MMU	See mmu-layout.txt.
SCR1	MMU	See mmu-layout.txt.
SCR2	MMU	Save for EAR0 (destroyed by icache insns in debug mode)
SCR3	MMU	Save for GR31 during debug exceptions
DAMR/IAMR	NOMMU	Fixed memory protection layout.
DAMR/IAMR	MMU	See mmu-layout.txt.

Certain registers are also used or modified across function calls:

REGISTER	CALL	RETURN
GR0	Fixed Zero	–
GR2	Function call frame pointer	
GR3	Special	Preserved
GR3–GR7	–	Clobbered
GR8	Function call arg #1	Return value (or clobbered)
GR9	Function call arg #2	Return value MSW (or clobbered)
GR10–GR13	Function call arg #3–#6	Clobbered
GR14	–	Clobbered
GR15–GR16	Special	Preserved
GR17–GR27	–	Preserved
GR28–GR31	Special	Only accessed explicitly
LR	Return address after CALL	Clobbered
CCR/CCCR	–	Mostly Clobbered

INTERNAL DEBUG-MODE REGISTER ABI

This is the same as the kernel-mode register ABI for functions calls. The difference is that in debug-mode there's a different stack and a different exception frame. Almost all the global registers from kernel-mode (including the stack pointer) may be changed.

REGISTER	FLAVOUR	USE
GR1		Debug stack pointer
GR16		GP-Rel base register for small data
GR31		Current debug exception frame pointer (__debug_frame)
SCR3	MMU	Saved value of GR31

Note that debug mode is able to interfere with the kernel's emulated atomic ops, so it must be exceedingly careful not to do any that would interact with the main kernel in this regard. Hence the debug mode code (gdbstub) is almost completely self-contained. The only external code used is the printf family of functions.

Furthermore, break.S is so complicated because single-step mode does not switch off on entry to an exception. That means unless manually disabled, single-stepping will blithely go on stepping into things like interrupts. See gdbstub.txt for more information.

VIRTUAL INTERRUPT HANDLING

Because accesses to the PSR is so slow, and to disable interrupts we have to access it twice (once to read and once to write), we don't actually disable interrupts at all if we don't have to. What we do instead is use the ICC2 condition code flags to note virtual disablement, such that if we then do take an interrupt, we note the flag, really disable interrupts, set another flag and resume execution at the point the interrupt happened. Setting condition flags as a side effect of an arithmetic or logical instruction is really fast. This use of the ICC2 only occurs within the kernel – it does not affect userspace.

The flags we use are:

(*) CCR.ICC2.Z [Zero flag]

Set to virtually disable interrupts, clear when interrupts are virtually enabled. Can be modified by logical instructions without affecting the Carry flag.

(*) CCR.ICC2.C [Carry flag]

Clear to indicate hardware interrupts are really disabled, set otherwise.

What happens is this:

(1) Normal kernel-mode operation.

ICC2.Z is 0, ICC2.C is 1.

(2) An interrupt occurs. The exception prologue examines ICC2.Z and determines that nothing needs doing. This is done simply with an unlikely BEQ instruction.

(3) The interrupts are disabled (local_irq_disable)

ICC2.Z is set to 1.

(4) If interrupts were then re-enabled (local_irq_enable):

ICC2.Z would be set to 0.

A TIHI #2 instruction (trap #2 if condition HI - Z==0 && C==0) would be used to trap if interrupts were now virtually enabled, but physically disabled – which they're not, so the trap isn't taken. The kernel would then be back to state (1).

(5) An interrupt occurs. The exception prologue examines ICC2.Z and determines that the interrupt shouldn't actually have happened. It jumps aside, and there disabled interrupts by setting PSR.PIL to 14 and then it clears ICC2.C.

(6) If interrupts were then saved and disabled again (local_irq_save):

ICC2.Z would be shifted into the save variable and masked off (giving a 1).

ICC2.Z would then be set to 1 (thus unchanged), and ICC2.C would be unaffected (ie: 0).

- (7) If interrupts were then restored from state (6) (local_irq_restore):

ICC2.Z would be set to indicate the result of XOR'ing the saved value (ie: 1) with 1, which gives a result of 0 – thus leaving ICC2.Z set.

ICC2.C would remain unaffected (ie: 0).

A TIHI #2 instruction would be used to again assay the current state, but this would do nothing as Z==1.

- (8) If interrupts were then enabled (local_irq_enable):

ICC2.Z would be cleared. ICC2.C would be left unaffected. Both flags would now be 0.

A TIHI #2 instruction again issued to assay the current state would then trap as both Z==0 [interrupts virtually enabled] and C==0 [interrupts really disabled] would then be true.

- (9) The trap #2 handler would simply enable hardware interrupts (set PSR.PIL to 0), set ICC2.C to 1 and return.

- (10) Immediately upon returning, the pending interrupt would be taken.

- (11) The interrupt handler would take the path of actually processing the interrupt (ICC2.Z is clear, BEQ fails as per step (2)).

- (12) The interrupt handler would then set ICC2.C to 1 since hardware interrupts are definitely enabled – or else the kernel wouldn't be here.

- (13) On return from the interrupt handler, things would be back to state (1).

This trap (#2) is only available in kernel mode. In user mode it will result in SIGILL.