What's New#

QNX takes advantage of new ARMv6 features to remove some of the 'quirks' in the ARM implementation:

- new atomic instructions (ldrex/strex)
- new MMU architecture that implements a physically tagged cache
- new MMU protection that means the kernel can't write to user read-only pages
- architectural support for floating point via the VFP instruction set

The new ARMv6 support is implemented in:

- libstartup.a to initialise and configure an ARMv6 cpu
- procnto-v6 that implements ARMv6 specific kernel and memmgr support
- libc.a/libc.so that takes advantage of ldrex/strex for atomic operations

libstartup#

hardware/startup/lib/arm contains the generic support for configuring and initialising the cpu and data structures required for procnto-v6:

- set up the MMU and page tables to use ARMv6 operation (VMSAv6 in ARMARM speak)
- detect and initialise the VFP

The ARMv6 support is mainly implemented in armv_setup_v6.c:

- detect and enable coprocessor access to the VFP
- set up the TTBCR for a 2GB user address space and 2GB system address space
- set system page flags so procnto-v6 and libc know we're on an ARMv6 cpu

Each ARMv6 chip-specific setup function should call armv_setup_v6() to do this generic initialisation in addition to whatever chip specific stuff is needed. See the following for details for each processor:

- armv_setup_1136.c initialises an ARM1136 cpu
- armv_setup_1176.c initialises an ARM1176 cpu
- armv_setup_mpll.c initialises an ARM11 MPcore cpu

Page tables are set up using one of the following:

- armv_pte_v6wb.c defines descriptors for write-back caching
- armv_pte_v6mp.c defines descriptors for an MPcore system

The particular ones to use are specified by the chip-specific configuration:

- armv_chip_1136.c pulls together the pieces required for an ARM1136 cpu
- armv_chip_1176.c pulls together the pieces required for an ARM1176 cpu
- armv_chip_mpll.c pulls together the pieces required for an MPcore cpu

procnto-v6#

This kernel variant implements support for ARMv6 processors:

- kernel specific support and optimisations
- memory manager for the VMSAv6 MMU architecture

Most of the code is shared with the regular (non-ARMv6) procnto. ARMv6 specific code is in:

- VARIANT_v6 conditional code scattered throughout the source
- memmgr/arm/v6
- ker/arm/v6

The new access protections mean privileged access to user read-only pages no longer needs special handling:

- kercpu.h: WR_PROBE_OPT is no longer required
- cpu_debug.c: breakpoints need to be manipulated through an alias
- xfer_copy.S: doesn't need a special copy routine for user pages
- nano_xfer_cpy.c doesn't need special handling for user memory

kernel.S implements some ARMv6 specific support and optimisations:

- exception entry uses cps to switch modes on exception entry
- we don't need to deal with the Xscale CP0 state
- we don't need to use the FCSE PID and associated MMU domain register code
- we need to clear the exclusive monitor on kernel exit

ker/arm/v6 implements atomic operations using ldrex/strex

The memory manager code is radically different to the old implementation:

- user address space is now 2GB instead of 32MB
- no special behaviour for shm_ctl()

The new code is in memmgr/arm/v6. Stay tuned for the gory details on how this works...

VFP Support#

Basic support for VFP hardware is implemented in the kernel:

- kernel.S has hooks for driving lazy context switching in ker_exit
- cpu_undef.c has hooks for dealing with VFP exceptions
- vfp.c implements the saving/restoring of VFP state

The kernel assumes startup will set CPU_FLAG_FPU if it detects a VFP. If this is set, then vfp_init() is called to set up the VFP support.

This support provides:

- lazy context switching of VFP state
- support for only RunFast mode (ie. no trapped FP exceptions)

There are sufficient hooks to allow an fpemu.so implementation to service trapped FP exceptions, but that mode of operation is not really suited to a realtime embedded system.

The lazy context switch support basically works as follows:

- initially, the VFP is disabled

- when a thread executes its first VFP instruction, it causes an undefined instruction exception
- the exception handling code sets _NTO_ATF_FPUSAVE_ALLOC to indicate that this thread needs to allocate a save area for its VFP context
- on return through __ker_exit, fpusave_alloc() is called to allocate the save area
- the faulting instruction is restarted, but since the FPU is still disabled another undefined instruction exception occurs
- this time, the handler will perform a VFP context switch to load the thread's (initial) VFP context and enable the VFP

The kernel keeps track of the 'active' VFP context in actives_fpu. On return to user mode, __ker_exit checks if this matches the thread we are returning to - if it doesn't match, the VFP is disabled so that if the thread uses the VFP it will cause an undefined instruction exception:

- if the thread has a VFP save area, a VFP context switch is performed
- if it has no VFP save area, one is allocated via _NTO_ATF_FPU_SAVEALLOC

libc#

The only significant change to libc is to use the new ldrex/strex instructions for atomic operations if ARM_CPU_FLAG_V6 is set.

See the code in lib/c/atomic/arm