The odd kid on the block

or: to boldly run ARM like no one did before

Martin Husemann
martin@NetBSD.org

Abstract

Modern ARM SoCs offer bi-endian support: the CPU can switch between little and big endian mode. Similar to the old hacker phrase "all the world's a VAX" of course everyone (and most software) expects ARM processors to be little endian.

Matt Thomas realized this would be an interesting challenge for the NetBSD portability mantra and added support for this mode to NetBSD/evbarm. When the NetBSD foundation offered me a CubieTruck[1] for my regular "test runs on strange hardware" lab[2], I decided to use the big endian kernel and see what fallout happens.

This paper describes the problems I found and the solutions chosen.

Of course in the end it is not the identification of a bug that counts, but its elimination. The whole process so far was a big success: currently the automatic test runs show a tie between armv5-little endian, armv7-big endian, and sparc64 at the top rank of all architectures with only one unexpected failure in NetBSD-current[3] and zero on the netbsd-7 branch heading towards the NetBSD 7 release.

1. Pre-Installation Issues

When I got the board, many peripherals did not work. Luckily Jared McNeill and Matt Thomas fixed most missing drivers quickly, but I ended up doing the ethernet and nand driver (the latter is still work in progress as of the time of this writing).

Ethernet was essential, so creating this driver was the first thing to do. Initially no documentation was available besides u-boot sources and a Linux driver, which as usual contained various magic numbers without any explanation. The A20 SoC implements an IP core from Synopsis Designware, which is also used (apparently in slightly different or newer versions) in various other chips – and later we have been able to find documentation for some of those.

In retrospect, this phase took the most time.

2. The ARM BE8 ELF Image Format

Modern ARM CPUs support big endian mode in a new way called "be8" (the legacy big endian mode in older implementations has been renamed to "be32").

The main features of the new big-endian mode are

- Easily switchable at runtime (even in unprivileged/user mode code)
- Instruction encoding is always little endian

The ELF object format[4] for be32 has been "reused" for be8, but this means all instructions are encoded in the wrong (big endian) byte order. To allow fixing this, the ABI defines special symbols marking the beginning of 32bit arm code (`$a`, `$a.*`), 16 bit thumb code (`$t`, `$t.*`) or other data (`$d`, `$d.*`). Usually the linker (when invoked with -be8) swaps the instructions accordingly when generating the final executable image.

This process leaves quite a few chances for Murphy's law to hit- and it did.

You may imagine my confusion when first looking at a core dump where instructions disassembled differently (due to different endianness decoding of instructions) and made no sense at all, but the corresponding .o file disassembled correctly into reasonable instructions, - while the CPU seemed to agree with the swapped (completely bogus) code - which caused an exception (PR 49337[1]).

[1] PR port-arm/49337: __popcountsi2() triggers illegal instruction; http://gnats.netbsd.org/49337

3. Big endian Problems found

The big endian testing revealed the following problems:

3.1. Early serial console code had byte order issues

To allow sending debug output to the serial port before attaching any drivers, a simple polled "early console" is setup (struct consdev awin_earlycons[2]) and two simple functions send and receive characters to/from it. Since there is no suitable bus_space to available yet, this functions access the hardware registers directly – and this access was done in host byte order. A noticeable delay was caused by this, as the hardware never reported TXRDY. The delay was followed by some paragraphs of garbage output, until finally the kernel console driver attached and normal boot messages followed.

After identifying the issue, the obvious fix was easy: insert le32toh() calls, like:

```
while ((le32toh(uart_base[com_lsr])
      && LSR_TXRDY) == 0 && --timo > 0)
           ;
```

3.2. MMC driver used host endianness in DMA descriptors

A similar error happened in the awin_mmc.c driver. While standard access to hardware registers automatically deals with host endianness at the bus_space layer, all DMA descriptors passed to the device or received from it need special care to ensure the device and the host CPU talk the same byte-order. This is easy to overlook, resulting in arbitrary DMA operations overwriting random memory and (in my case) the device locking up hard. But it also is easy to fix by inserting htole32[3] and le32toh calls (the latter were not needed in this case, since the device reports all status information via registers).

3.3. Kernel module loader did not support BE8 instruction swaps

During automatic test runs the kernel module loader is exercised. When first tried, this caused crashes, as reported in PR 49299[4].

The module loadable files are not finally linked (with -be8 option), but instead handled by the kernel object loader. The load had to be taught about the magic `$a` (and friends) symbols and do byte swapping post-load. This turned out to be quite simple:

The basic idea of the special marker symbols (as described above) is: when running on a BE8 CPU, you have to know which parts of `.text` (or other sections containing program code) are raw data, 32bit arm code, and 16bit thumb code. Once identified, you swap 32bits in the arm code sections, or 16bits in the thumb code. Only the start of the parts is marked with a "`$d`", "`$a`" or "`$t`" symbol (or numbered variants of it, like "`$a.1`").

We do the needed byte swapping in the kobj_machdep function, which is called right after a module has been loaded and relocations fixed up:

```
int
kobj_machdep(kobj_t ko, void *base,
    size_t size, bool load)
{

        if (load) {
#if __ARMEB__
                if (CPU_IS_ARMV7_P())
                        kobj_be8_fixup(ko);
#endif
```

Then we need a simple function to categorize symbols:

2 sys/arch/arm/allwinner/awin_board.c, lines 96-125
3 sys/arch/arm/allwinner/awin_mmc.c, lines 756ff
4 PR port-arm/49299: earmv7hfeb kernels can not load modules; http://gnats.netbsd.org/49299

```
/*
 * See ELF for the ARM Architecture,
   Section 4.5.5: Mapping Symbols
 * ARM reserves $a/$d/$t (and variants
   like $a.2) to mark start of
 * arm/thumb code sections to allow
   conversion from ARM32-EB to -BE8
 * format.
 */
static enum be8_magic_sym_type
be8_sym_type(const char *name, int info)
{
    if (ELF_ST_BIND(info) != STB_LOCAL)
            return Other;
    if (ELF_ST_TYPE(info) != STT_NOTYPE)
            return Other;
    if (name[0] != '$' || name[1] == '\0' ||
        (name[2] != '\0' && name[2] != '.'))
           return Other;

   switch (name[1]) {
   case 'a':
           return ArmStart;
   case 'd':
           return DataStart;
   case 't':
           return ThumbStart;
   default:
           return Other;
   }
}
```

We can then use the convenient ksyms_mod_foreach iterator function to find all relevant symbols, categorize them, count them or enter them into a list:

```
/*
 * Count all special relocations symbols
 */
ksyms_mod_foreach(ko->ko_name,
    be8_ksym_count, &relsym_cnt);
```

Here is the counting callback function as an example how the iterator callback looks:

```
static int
be8_ksym_count(const char *name,
    int symindex, void *value,
    uint32_t size,
    int info, void *cookie)
{
  size_t *res = cookie;
  enum be8_magic_sym_type t =
           be8_sym_type(name, info);

  if (t != Other)
     (*res)++;
  return 0;
}
```

Straight forward. I am leaving out the boring details of the other steps, you can check them at kobj_machdep.c[5]. We need another iterator callback function to enter the symbols into a pre-allocated array, then a comparator function to sort symbols by address via kheapsort, and finally just run through the sorted array, swapping 16 or 32 bytes – or not at all, depending on the last symbol marker type we have seen.

Since we inserted this all at the beginning of kobj_machdep(), cache coherency/flushing is handled later anyway.

After this change, module loading worked – but showed another issue (see PR 49396[6]) on machines with large memory using a direct mapped kernel segment: modules may be loaded too far away from kernel text – but this is not BE8 specific. It will be solved by adjusting compiler options (-mlong-calls or similar), or changing the location where modules are loaded (like on other architectures). A combination of MMU details and possible branch distance range make this not as easy as it sounds, this will need further investigation.

3.4. libgcc was miscompiled, so -eb8 swaps did not work at link time

Very similar to the kernel loader issue, and also very confusing, this issue came up early: late in the boot process, `fc-cache(1)` is run to update the X font cache. On the CubieTruck this program invocation crashed, resulting in an error log from the rc scripts.

This was the first time erroneously swapped instruction encodings came up and I did not have my homework

[5] sys/arch/arm/arm32/kobj_machdep.c, lines 210-394
[6] PR kern/49396: reproducable panic on puffs; http://gnats.netbsd.org/49396

(learning the details about BE8) done, so the original bug report was fuzzy: PR 49337[7].

Later analysis showed: the magic local symbols marking 32bit code sections (and thus instructing the linker to swap for be8 mode) had been removed, resulting in byte swapped code for all gcc intrinsic functions.

In a standard build, NetBSD does not use the gcc provided build infrastructure to build libgcc, instead all “configury” is done upfront during a step called “mknative”, and the resulting makefile fragments and header files are then committed to the NetBSD tree. This guarantees reproducible build results and allows for easier cross builds. The Net-BSD makefiles driving the final build of libgcc force visibility of all libgcc symbols to be “hidden” and play some tricks that involved a strip and ld –r step. Unfortunately strip does like it is told – and strips all local symbols, including the EABI reserved byte swapping markers `$d`, `$a` and `$t`. The final link step creating libgcc_s.so later could not identify the BE8 instructions, so did not swap anything, resulting in reversed byte order instructions in the shared library.

Once identified, a fix was simple: strip got replaced by a slightly cleverer objcopy invocation, preserving the special `$`-symbols.

3.5. "run" in gdb did not work

When automatic test runs result in failures, it is often useful to run the test program inside gdb and set breakpoints, so state can be examined before the failure happens. Unfortunately this did not work on BE8 ARM, as reported in PR 49445[8].

Now how do you debug gdb, if you cannot run gdb in gdb? Even if you can run gdb in gdb it is a bit scary, recursive and insane. Given the history so far of byte swapping (or lack of) operations for BE8 code, it was likely that this was another instance. Code reading in gdb showed that newer ARM targets used two different markers for endianness in their `struct gdbarch_info`: `byte_order` for general operations and `byte_order_for_code` for instructions. However, the NetBSD specific code in armnbsd-tdep.c[9] had not been updated.

The code in question selects the encoding for breakpoint instructions. Gdb automatically inserts a breakpoint in the runtime linker, ld.elf_so, which gets hit after loading any new shared library. Due to the wrong endianness of the breakpoint encoding, the ld.elf_so code was modified but the breakpoint not hit. The correct breakpoint encoding is an undefined instruction:

```
110000e6   ; <UNDEFINED> instruction:
                0xe6000011
```

but with reversed byte order it is a conditional signed multiply:

```
e6000011   smlattne   r0, r6, r0, r0
```

So instead of causing an invalid instruction trap, which would have been trapped by gdb, the code modified register r0 and continued to run into unrelated code - since the “breakpoint” replaced a return instruction, as that is the only instruction in `_rtld_debug_state`:

```
(gdb) x/3i _rtld_debug_state
   0x21e8 <_rtld_debug_state>:  bx      lr
   0x21ec <_rtld_objlist_clear>:
                           mov     r12, sp
   0x21f0 <_rtld_objlist_clear+4>:
         push    {r3, r4, r11, r12, lr, pc}
```

The sole purpose of the `_rtld_debug_state()`[10] function is to provide a hook for debuggers. But now execution continued in `_rtld_objlist_clear`, with bogus arguments.

Fixing the NetBSD gdb target code to use the new `byte_order_for_code` field resolved this issue.

4. Other problems found

Not all issues hit were caused by byte order differences, but just being different.

4.1. C++ exception unwinding did not work

When compiled with gcc (we offer clang/llvm as an option on ARM), NetBSD uses a strange combination of libstdc++ from gcc and the native libunwind (based on LLVM runtime code).

The automatic testing framework (ATF) is written in C++, and one of its internal tests cases reliably failed with a segmentation violation during exception unwinding, as reported in PR 49444[11].

Exceptions are evil, and unwinding them is close to a black art. The toolchain creating binaries produces additional data, called CFI (call frame information)

[7] PR port-arm/49337: __popcountsi2() triggers illegal instruction; http://gnats.netbsd.org/49337
[8] PR toolchain/49445: gdb can't "run" programs on BE8 arm; http://gnats.netbsd.org/49445
[9] external/gpl3/gdb/dist/gdb/armnbsd-tdep.c, line 45ff

[10] libexec/ld.elf_so/rtld.c, lines 1431ff
[11] PR lib/49444: c++ exception unwinding broken on BE8 arm; http://gnats.netbsd.org/49444

encoded in DWARF (the ELF debug information format) and stored in the `.eh_frame` section of the binary. To save space, parts of the data are compressed as byte code for a special finite state machine.

To unwind an exception, the current PC is looked up in the unwind info, and then instructions there are followed how to get to the callers frame and the process is repeated, frame by frame.

The `objdump(1)` utility can dump the DWARF data in readable form, but the output looked wrong, especially the PC ranges given have an offset which has not been explained to me so far, maybe it is a bug in objcopy or BFD, but at first sight it seemed to point at gcc creating bogus unwind info. Comparing with very similar output for the (working) little endian ARM case ruled this out.

There is another tool in the binutils collection, `readelf(1)`, which does not use the object format agnostic libbfd, but is hardcoded for ELF. It can also decode the CFI data, and displays it correctly – so gcc was off the hook.

Adding annotations to the unwind library showed this was a bug in a binary search, when the target PC was in the last block of data, but not the first address of that range. A slightly different overall memory mapping triggered this, the little endian version was just lucky.

4.2. FPU exceptions missing

The cortex-7 CPUs do not implement FPU-exceptions, so a few test programs had to be adjusted. The trick is to test whether the FP_X_INV bit is sticky in the FPU exception mask:

```
#elif defined(__arm__) && !__SOFTFP__
    /*
     * Some NEON fpus do not implement IEEE
       exception handling,
     * skip these tests if running on them
       and compiled for
     * hard float.
    */
    if (0 == fpsetmask(fpsetmask(FP_X_INV)))
       atf_tc_skip("FPU does not implement"
                   " exception handling");
#endif
```

This might look like a copy & paste error at quick glance, but it is correct: the inner fpsetmask sets the exception enable bit for invalid operations and returns the previous mask. The outer fpsetmask restores the previous mask and returns the FP_X_INV from the inner call if the FPU implements it, or zero if not.

4.3. Unaligned access works

Similarly, ARM since version 6 does not require the same strict alignment as older versions. To help user land programs to query this property, a sysctl (machdep.unaligned_sigbus) has been implemented and is now used in tests for alignment exceptions.

The same sysctl already existed on the Alpha architecture (where it even is user settable, so the behavior can be changed at runtime and the kernel fixes access in trap handlers). This is not possible on ARM, but adding the sysctl value read-only made modifications of the test code very simple:

```
#if defined(__alpha__) || defined(__arm__)
    int rv, val;
    size_t len = sizeof(val);
    rv = sysctlbyname(
       "machdep.unaligned_sigbus",
       &val, &len, NULL, 0);
    ATF_REQUIRE(rv == 0);
    if (val == 0)
        atf_tc_skip("No SIGBUS signal for"
                    " unaligned accesses");
#endif
```

The only necessary user land change here was modifying the #if instruction to include ARM. The kernel side implementation is trivial as well: the current CPU is tested for supporting ARMv6 or ARMv7 instructions:

```
cpu_unaligned_sigbus =
    !CPU_IS_ARMV6_P() && !CPU_IS_ARMV7_P();
```

5. Conclusion

In retrospect the issues found and fixed were less than expected. Most time was spent on typical problems when bringing up new hardware. After basic testing works, the automatic tests (via ATF) proved very valuable again, but some issues did not get noticed – so there is further room for improvement.

Only small parts of pkgsrc (3rd party software) have been tested so far, but with surprisingly few problems – autoconfig (and alternatives) do a good job, few assumptions are hardcoded nowadays.

The big test will follow: getting Firefox running in big endian mode on arm. It uses lots of assembly code (ARM is one of the best supported platforms) and has JIT compilers, so this sure will be another fun project.

Compared to fifteen years ago when my other favorite odd architecture (of course that is sparc64) became popular in NetBSD and people started throwing third party software from pkgsrc (and even X) at it, the whole experience was extremely smooth and painless. Maybe we all ARE improving, after all and software gets better (or at least more portable).

The not endian related issues showing up are a sign that diversity (especially in testing environments) always pays. For example the libunwind bug would have been extremely hard to track down if it had not (out of pure luck) manifested itself in a simple automatic test case.

6. Acknowledgments

I would like to thank all the people who helped with the CubieTruck, especially Matt Thomas, Nick Hudson, and Jared McNeill. Also many thanks go to Jörg Sonnenberger for fixing the nasty libunwind bug.

The NetBSD Foundation, Inc. funded my CubieTruck board, which made this work possible.

Finally many thanks to the AsiaBSDCon program committee and organizers for providing the conference and accepting my talk.

7. References

[1] http://cubieboard.org/

[2] Log data from automatic test runs: http://releng.netbsd.org/test-results.html

[3] CubieTruck test results page: http://www.netbsd.org/~martin/evbearmv7hf-atf/

[4] ELF for the ARM® Architecture: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044e/IHI0044E_aaelf.pdf

[5] Martin Husemann, *ARM multiprocessor support*, with CubieTruck images: http://blog.netbsd.org/tnf/entry/working_arm_multiprocessor_support

[6] NetBSD/evbarm on Allwinner Technology SoCs: http://wiki.netbsd.org/ports/evbarm/allwinner/