QNX Best Practices#

1. Introduction

The purpose of this document is to establish the best practices for QNX software development.
QNX Internal Reference: QNX Coding Standard - QMS0021

2. QNX Best Practices

2.1. GPL/Third Party Code

Under no circumstance should GPL (Linux) code be used. When in doubt talk to the QNX Legal Department.

2.2. Portablity

When implementing source, you should always try to write the code in a manner that is OS and CPU independent. If it's not possible to do that, the next best thing is to implement a function level API that hides the difference. The main source code calls this function that will be re-implemented individually for each OS or CPU. The source code for the function will live in separate source files, lower down in the source hierarchy. If a functional interface is not appropriate for efficiency or other reasons, the next best thing is to use a function-like macro that's defined in a header file to hide the difference. ""Only as a very last resort should a conditional compilation directive be placed in a source file"".

When you add a #ifdef chain to something, always terminate it with a:

```
…
#else
    #error THING not supported
#endif
```

True, it means that the next guy might get errors when he first compiles something, but better to make him look at the code and decide what to do than let it silently go through and spend a day or so trying to figure out what's wrong. Taking this further, when you're implementing something, if you know that it won't work for some given condition and you can test for that condition, put in an #error report.

Drivers for PCI devices must support big and little endian architectures.

- Refer to /usr/include/gulliver.h for endian conversion macros.
- Refer to /usr/include/hw/inout.h for endian aware I/O routines

  ie inle32().

2.3. Function Prototypes

Use function prototypes. There is no excuse for not using them. You should define a function prototype only once in the entire source, in one header file. Defining multiple prototypes defeats the purpose of having a function prototype in the first place.

2.4. Coding for Reliability

Even if your algorithms and code implement all of a specification or requirements document, your code can and will be subject to incorrect, out-of-specification, or malicious data. In order for your code to survive, you must check for out-of-specification or unexpected data

values and act accordingly and reliably when such input values are passed to your code. Validate data at "external" input points and assume proper input for "internal" points. Always check return values.

The assert() statement is enabled only during debug builds of the software. This statement can be left in the release source code if it is not distracting such that if need be experimental debug versions can be distributed that highlight error conditions quickly.

2.5. Case statements

Include default cases on all switch statements, and do not allow an unhandled value to fall through into the code that follows the switch statement. Also, make note of when you intend a case of a switch statement to fall through into the next case selector. For example:

```
switch (number) {
case 0:
    printf( "You entered zero. I assume you meant 1.\n" );
    /* Fall through */
case 1:
    dostuff( );
    break;
case 2:
    …
    break;
default:
    printf( "Unknown case\n" );
    break;
}
```

2.6. IF…Else statements

Always bracket if/else statements. Your code is not likely to remain unchanged forever and adding brackets ensures that the logic of your code is evident. Your code is like For example:

```
if (something) {
    do stuff…
}
```

Avoid single line statements such as:

```
if (something) { do stuff }
```

These type of statements make it impossible to plant breakpoints on the inner code in the debugger and may result in misleading results for profilers and memory analysis tools.

2.7. Goto Statement

Avoid using the goto statement. However in some case it can be used to preform error cleanup for functions that allocate a lot of resources.

2.8. Typecasting

Do not use gratuitous typecasting. Functions that return void * (known as an opaque type) need no typecasting of the returned pointer. Doing so defeats the whole purpose of declaring the function with an opaque type.

2.9. Obscure C Features

Avoid using obscure C language features, such as the "," operator. If you are in doubt about a feature's "obscureness" ask your fellow developers about the feature. If they are not intimately familiar with the language feature then it may be questionable to use it.

The maintenance improvement from better source code readability far outweighs any optimizations that might have been achieved with older compilers.

2.10. Ensuring Correct Results

Do not depend on nonobvious associativity and side effects for correct results. It is unwise to rely on the order of evaluation of side effects performed on variables passed to functions, and it is extremely unwise to use side effects in the invocation of a macro.

2.11. Static Class

Use the static storage class to reduce the scope of variables and functions. Do not make any variable or function an extern unless it is used outside the file in which it is defined.

2.12. Const Qualifier

Use const type qualifiers to allow the compiler to enforce read-only declarations of read-only global storage and variables that should not change at run time.

In addition to declaring initialized storage to be const, you should also declare and define the read-only parameters of functions to be const.

2.13. Conversion from Signed to Unsigned Types

Pay attention to the conversion between signed and unsigned types, in both directions. This is one area where the transition from K&R C to ANSI C occasionally surprises people.

2.14. Enumerated Types and #defines

When possible, use ordinals declared using enum, especially for arguments to switch statements. GCC will warn you about unhandled values, a warning you will not receive when using arguments of type int or unsigned integers.

If you are creating a list of #define macros for enumerated constants, consider whether you can use a real enumerated type rather than the #define idiom. You can assign explicit values to enumerated type names if required, as shown in this example:

```
enum {
    first_value = 1,
    next_value  = 2,
    last_value  = 3
};
```

If you must use the #define idiom instead of enumerated types, write the #defines so they are self-indexing if that is the intended use. For example:

```
#define EXAMPLE_ITEM_ONE        (1)
#define EXAMPLE_ITEM_TWO        (EXAMPLE_ITEM_ONE+1)
#define EXAMPLE_ITEM_THREE      (EXAMPLE_ITEM_TWO+1)
```

If the #define idiom is being used for bitfields or field masks, align the entries so that errors can be quickly identified. For example:

```
#define FLAG_A  0x00000001
#define FLAG_B  0x00000002
#define FLAG_C  0x00000004
```

is prefferable over:

```
#define FLAG_A  1
#define FLAG_B  2
#define FLAG_C  4
```

One reason to favour the #define idiom over an enumerated type would be that the constant definitions can be processed by both the C preprocessor and other, non-C, macro languages.

2.15 Floating-Point Operations

Many of our target platforms have no floating-point hardware and the operations will have to use the emulator. Do not use floating-point operations in: drivers, driver utilities, libc, kernel. Consider using fixed point math.

2.16 Coding for Performance

Use "natural" sizes. If a target architecture accesses a 16-bit memory location faster than it accesses an 8-bit location, consider declaring your storage to be a 16-bit large area. You must then weigh this increase in size against other factors, such as cache hit ratios and cache line packing.

2.17 Naming

When giving names to functions/variables/macros that are CPU/device dependent, always prefix the name with something that identifies the CPU/device that the symbol is for. For example, anything that is specific to an Intel x86 processor is always prefixed with "x86" or "X86" as appropriate. This serves two purposes. First, it makes sure that we don't have any naming conflicts between CPU/device's.

Second, it makes it obvious when the source code is CPU/device dependent. This prefix naming continues to lower levels as well. For example, if you're defining the bits on the ABC register of the XYZ processor, all the definitions should be prefixed with XYZ_ABC. Examples:

```
MIPS_SREG_BEV        - MIPS status register BEV bit
PPC_FPSCR_VE         - PPC floating point status/control reg VE bit
PPC403_BESR_DSES     - PPC 403 bus error syndrome register DSES bit
S2681_CSR_S1BAUD75   - Signetics 2681 serial chip clock select reg
```

Also, separate the definitions into appropriately named files. For example, all definitions about generic PPC CPU registers appear in <ppc/cpu.h>. Definitions specific to the PPC 403 chip appear in <ppc/403cpu.h>. Note that this is not a hard and fast rule. If there are only one or two additional registers/bits for a particular processor, it might be more efficient to just include the definitions in the main file (but still using the more explicit prefix!). Use your own judgment on this.

All function/variable names should be lowercase with the following properties:

Legibility
: Names should be meaningful and have a maximum length of 15 characters.

Short Names
: Single letter names should not be used except for counters (i, j, k).

Separating Names

Use underscores to separate names.

2.17.1 Namespaces

Reserved Namespace

The 'C' standard requires that all system-specific macros be part of the reserved namespace. All names which begin with two under-scores, or an underscore and a capital letter, are reserved for compiler and C library to use as they wish.

Libraries should endeavour to reserve their own chunks of namespace to ensure that accidental name collisions are minimized.

User Namespace

The user namespace must meet the following requirements:

- It does not begin with an underscore.
- It is not a keyword in the language
- It is not reserved for the ANSI library.

3. Driver issues:

3.1 Interrupts

Handlers should be as short/fast as possible. Remember that QNX is an RTOS and customers expect a small interrupt latency. Customers measure the latency, so your evils will be found.

Drivers must support sharing interrupts with other devices.

Common problems:

- Attaching to an interrupt before the driver is able to properly service it.
- Not disabling the device interrupt when the driver terminates.

3.2 Process Time Handlers

Most QNX drivers handle interrupts at process time by using InterruptAttachEvent(). The handler should clear the device interrupt as fast as possible and call InterruptUnmask().

3.3 Interrupt Time Handlers

The handler should clear or mask the device interrupt as fast as possible.

3.4 Bitfields

Don't use bitfields when attempting to define a structure that something external will see (device register layout, etc). Not all compilers will lay out bitfields in the same way. Use of explicit constants with and/or'ing is much more portable and will likely give you better code generation anyway.

3.5 Register Descriptions

All register offsets and contents must have Macro definitions. Magic numbers are not allowed.

Busy waits must be bounded. Never rely on the hardware performing as documented. Use a counter to break out of the loop and follow up with error recovery. For example:

```
int reset_device(int iobase) {
    out8( iobase + DEVICE_CTRL, DEVICE_CTRL_RESET );
    for (retry = MAX_RETRIES; retry; retry--) {
        if ( in8( iobase + DEVICE_CTRL ) & DEVICE_CTRL_RESET_CMPLT ) {
            return EOK;
        }
    }
    // reset failed
        return ENODEV;
    }
```

3.6 Comments

Always document hardware anomalies.

3.7 Address Translation

Drivers should never assume a 1 to 1 virtual/physical address mapping.

PCI drivers should never use addresses read from PCI configuration space. The pci server is aware of the translation being used and returns it via the pci_dev_info structure (see CpuBaseAddress[x] members).

The mmap_device_io and mmap_device_memory functions should be used to map registers into the drivers address space. The mapped memory should be declared volatile.

3.8 Busmaster

PCI drivers should never assume a 1 to 1 address mapping for bus-mastering. The pci server is aware of the translation being used and returns it via the pci_dev_info struct (see CpuBmstrTranslation member). The translation must be applied to the physical address before it is given to the busmaster engine.

3.9 DMA/Busmaster Safe Memory

Never use memory returned by malloc for DMA tranfers (see mmap).

3.10 Performance

Drivers should avoid calling mem_offset/mem_offset64 repeatedly since it is a message pass to procnto. It is better to cache the physical address returned. This said, don't go wild with caching since it mandates that the memory be locked in place.

3.11 Threads

Most QNX drivers are multi-threaded therefore critical data structures and hardware registers must be protected with mutexes or semaphores. When drivers have a real interrupt handler spin locks can be used. Do not disable interrupts for protection.

3.12 Shutdown/Power Management

Drivers should perform the following:

- Disable device interrupt (as applicable)-
- Put chip in a powered-down state (as applicable)

- Free any system resources

4. Libc issues:

5. Tool issues:

6. Kernel issues: