Generic bitfield packing and unpacking functions

Problem statement

When working with hardware, one has to choose between several approaches of interfacing with it. One can memory-map a pointer to a carefully crafted struct over the hardware device's memory region, and access its fields as struct members (potentially declared as bitfields). But writing code this way would make it less portable, due to potential endianness mismatches between the CPU and the hardware device. Additionally, one has to pay close attention when translating register definitions from the hardware documentation into bit field indices for the structs. Also, some hardware (typically networking equipment) tends to group its register fields in ways that violate any reasonable word boundaries (sometimes even 64 bit ones). This creates the inconvenience of having to define "high" and "low" portions of register fields within the struct. A more robust alternative to struct field definitions would be to extract the required fields by shifting the appropriate number of bits. But this would still not protect from endianness mismatches, except if all memory accesses were performed byte-by-byte. Also the code can easily get cluttered, and the high-level idea might get lost among the many bit shifts required. Many drivers take the bit-shifting approach and then attempt to reduce the clutter with tailored macros, but more often than not these macros take shortcuts that still prevent the code from being truly portable.

The solution

This API deals with 2 basic operations:

- Packing a CPU-usable number into a memory buffer (with hardware constraints/quirks)
- Unpacking a memory buffer (which has hardware constraints/quirks) into a CPU-usable number.

The API offers an abstraction over said hardware constraints and quirks, over CPU endianness and therefore between possible mismatches between the two.

The basic unit of these API functions is the u64. From the CPU's perspective, bit 63 always means bit offset 7 of byte 7, albeit only logically. The question is: where do we lay this bit out in memory?

The following examples cover the memory layout of a packed u64 field. The byte offsets in the packed buffer are always implicitly 0, 1, ... 7. What the examples show is where the logical bytes and bits sit.

1. Normally (no quirks), we would do it like this:

```
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
7                       6                       5                       4
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
3                       2                       1                       0
```

That is, the MSByte (7) of the CPU-usable u64 sits at memory offset 0, and the LSByte (0) of the u64 sits at memory offset 7. This corresponds to what most folks would regard to as "big endian", where bit i corresponds to the number 2^i. This is also referred to in the code comments as "logical" notation.

2. If QUIRK_MSB_ON_THE_RIGHT is set, we do it like this:

```
56 57 58 59 60 61 62 63 48 49 50 51 52 53 54 55 40 41 42 43 44 45 46 47 32 33 34 35 36 37 38 39
7                       6                       5                       4
24 25 26 27 28 29 30 31 16 17 18 19 20 21 22 23  8  9 10 11 12 13 14 15  0  1  2  3  4  5  6  7
3                       2                       1                       0
```

That is, QUIRK_MSB_ON_THE_RIGHT does not affect byte positioning, but inverts bit offsets inside a byte.

3. If QUIRK_LITTLE_ENDIAN is set, we do it like this:

```
39 38 37 36 35 34 33 32 47 46 45 44 43 42 41 40 55 54 53 52 51 50 49 48 63 62 61 60 59 58 57 56
4                       5                       6                       7
7  6  5  4  3  2  1  0  15 14 13 12 11 10  9  8 23 22 21 20 19 18 17 16 31 30 29 28 27 26 25 24
0                       1                       2                       3
```

Therefore, QUIRK_LITTLE_ENDIAN means that inside the memory region, every byte from each 4-byte word is placed at its mirrored position compared to the boundary of that word.

4. If QUIRK_MSB_ON_THE_RIGHT and QUIRK_LITTLE_ENDIAN are both set, we do it like this:

```
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
4                       5                       6                       7
0  1  2  3  4  5  6  7  8   9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
0                       1                       2                       3
```

5. If just QUIRK_LSW32_IS_FIRST is set, we do it like this:

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
3                       2                       1                       0
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
```

```
 7                       6                       5                       4
```

In this case the 8 byte memory region is interpreted as follows: first 4 bytes correspond to the least significant 4-byte word, next 4 bytes to the more significant 4-byte word.

6. If QUIRK_LSW32_IS_FIRST and QUIRK_MSB_ON_THE_RIGHT are set, we do it like this:

```
24 25 26 27 28 29 30 31 16 17 18 19 20 21 22 23  8  9 10 11 12 13 14 15  0  1  2  3  4  5  6  7
3                       2                        1                       0
56 57 58 59 60 61 62 63 48 49 50 51 52 53 54 55 40 41 42 43 44 45 46 47 32 33 34 35 36 37 38 39
7                       6                        5                       4
```

7. If QUIRK_LSW32_IS_FIRST and QUIRK_LITTLE_ENDIAN are set, it looks like this:

```
7  6  5  4  3  2  1  0  15 14 13 12 11 10  9  8 23 22 21 20 19 18 17 16 31 30 29 28 27 26 25 24
0                       1                       2                       3
39 38 37 36 35 34 33 32 47 46 45 44 43 42 41 40 55 54 53 52 51 50 49 48 63 62 61 60 59 58 57 56
4                       5                       6                       7
```

8. If QUIRK_LSW32_IS_FIRST, QUIRK_LITTLE_ENDIAN and QUIRK_MSB_ON_THE_RIGHT are set, it looks like this:

```
0  1  2  3  4  5  6  7  8   9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
0                       1                       2                       3
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
4                       5                       6                       7
```

We always think of our offsets as if there were no quirk, and we translate them afterwards, before accessing the memory region.

Intended use

Drivers that opt to use this API first need to identify which of the above 3 quirk combinations (for a total of 8) match what the hardware documentation describes. Then they should wrap the packing() function, creating a new xxx_packing() that calls it using the proper QUIRK_* one-hot bits set.

The packing() function returns an int-encoded error code, which protects the programmer against incorrect API use. The errors are not expected to occur durring runtime, therefore it is reasonable for xxx_packing() to return void and simply swallow those errors. Optionally it can dump stack or print the error description.