Atomik Kernel API

Standard library types and macros

As Atomik is the lowest level multi-platform software in the operating system, it has to define its own types and libraries from scratch. Although many functions and types can be found in the user-level standard C library, most of them are particularized to the microkernel context.

The following document describes standard library types and macros widely used across the microkernel.

Types

```
#include <types.h>

typedef int8_t, uint8_t;
typedef int16_t, uint16_t;
typedef int32_t, uint32_t;
typedef int64_t, uint64_t;
```

These are the lowest level data types and represent signed (`intX_t`) and unsigned (`uintX_t`) integers of various lengths (8, 16, 32 and 64) in bits. Useful when dealing with architecture-dependant structures or binary files.

```
#include <types.h>

typedef uint8_t  BYTE;
typedef uint16_t WORD;
typedef uint32_t DWORD;
typedef uint64_t QWORD;
typedef int8_t   BOOL;
```

Windows-style low level data types intended to ease porting structures from Windows based code. The use of these types in new code is discouraged.

```
#include <types.h>

typedef unsigned int size_t;
```

This type describes the length of small-sized buffers. This type is not based in any low level type (it's just an unsigned integer) and its size depends on the compiler.

```
#include <types.h>

typedef void *physptr_t;
```

Generic pointer type used to describe physical addresses. It's usually found in virtual memory subsystem to differentiate between virtual and physical addresses and it's intended to make the code where both pointers are involved easier to read.

```
#include <types.h>

typedef memsize_t;
```

Unsigned integer long enough to measure arbitrary sizes in memory, from small buffers to the whole addressable space. Memory regions and big buffer sizes should be declared as this type.

```
#include <types.h>

typedef busword_t;
```

Unsigned integer with the size of the standard CPU word. It usually fits the biggest generic-purpose register and `memsize_t`, although this last requirement is not mandatory (for instance, in x86 with PAE, `memsize_t` must be bigger than `busword_t`).

Macros

```
#include <types.h>

#define PACKED
```

Macro used to pack elements within a structure, overriding any alignment restriction. Fields are stored in memory with no gaps among them, and the whole structure size equals to the sum of all individual sizes of each field.

```
#include <types.h>

#define ALIGNED(int size)
```

Forces any type to be aligned to **`size`** bytes. **`Size`** must be a power of two.

```
#include <types.h>

#define PACKED_ALIGNED(int size)
```

Forces a structure to be internally packed (as with `PACKED`) and makes its overall size aligned to **`size`**.

```
#include <types.h>

#define INLINE
```

Makes a function static and (if the compiler supports it) inline. Included for compatibility reasons for compilers without this feature. It expands to **`static`** in compilers with no inline function support and to **`static inline`** in compilers with inline function support.

```
#include <types.h>

#define __PAGE_BITS
```

Number of bits required by an integer to address any byte within a page. This value is architecture dependant. Under x86 platforms, `__PAGE_BITS` is 12 (4 KiB).

```
#include <types.h>

#define PAGE_SIZE (1 << __PAGE_BITS)
```

Size of a page (architecture dependant). Defined as the power of two of `__PAGE_BITS`. Under x86, this value is 4096.

```
#include <types.h>

#define PAGE_MASK (PAGE_SIZE - 1)
```

Bit mask with ones in the page-offset part of a memory address (least significant `__PAGE_BITS` bits) and zeros in the page-index (most significant remaining bits).

```
#include <types.h>

#define PAGE_OFFSET(void *addr)
```

Returns the page offset of a given address **`addr`**. This is equivalent to a bitwise AND operation with `PAGE_MASK`.

```
#include <types.h>

#define PAGE_START(void *addr)
```

Returns the address of the first byte of the page containing address **`addr`**. Equivlent to a bitwise AND operation with negated `PAGE_MASK`.

```
#include <types.h>

#define PAGE_NO(void *addr)
```

Returns the page index of the given address **addr**. Equivalent to a rightwards bitshift of __PAGE_BITS bits.

```
#include <types.h>

#define PAGE_ADDR(busword_t idx)
```

Returns the address of a page with the given index **idx.** Equivalent to a leftwards bitshift of __PAGE_BITS bits.

```
#include <types.h>

#define __UNITS(size_t x, size_t wrdsiz)
```

Returns the smallest number of **wrdsiz**-sized blocks required to store **x** bytes.

```
#include <types.h>

#define __ALIGN(size_t x, size_t wrdsiz)
```

Aligns the quantity **x** to **wrdsiz**. Equivalent to multiply __UNITS by **wrdsiz**.

```
#include <values.h>

#define FAILED(int action)
#define SUCCESS(int action)

#define FAILED_PTR(void *action)
#define SUCCESS_PTR(void *action)
```

Determines wheter the action **action** evaluates to a success or error value according to their return type. Under Atomik, most functions returning an integer error code return -1 (also defined as KERNEL_ERROR_VALUE) in case of error. In the other hand, functions returning a pointer mark their failure by returning NULL (also defined as KERNEL_INVALID_POINTER). FAILED/SUCCESS and FAILED_PTR/SUCCESS_PTR compare the evaluation of **action** to KERNEL_ERROR_VALUE and KERNEL_INVALID_POINTER respectively to determine whether that action has failed or not.

```
#include <util.h>

#define JOIN(x, y)
```

Joins two tokens **x** and **y** and expands to the resulting token **xy**. Useful for defining variable names depending on the values of one or more macros.

This macro also provides an easy way to define temporary variables inside other macros. If the following construction is found in line 38:

```
int JOIN(var, __LINE__);
```

It will expand to:

```
int var38;
```

```
#include <util.h>

#define STRINGIFY(x)
```

Turns the token or set of tokens **x** into the C string **"x"**. Useful when any sentence passed as a macro argument needs to be treated as a string by the code. Uses of STRINGIFY can be found in macros like ASSERT, which outputs a message with the unmet condition.

```
#include <util.h>

#define FAIL(const char *fmt, ...)
```

Outputs an error message to the current boot console and halts the microkernel. If interrupts are configured, a bugcheck interrupt is issued and a backtrace with a register dump is also produced.

```
#include <util.h>

#define CONSTRUCT_STRUCT(type, type *ptr)
```

Allocates memory for an object of type **type**, fills it with zeroes and stores a pointer to it inside **ptr**, which must be of type **type ***. Allocation is performed via **salloc**, therefore the underlying SLAB pool will depend on the current context (task or interrupt).

```
#include <util.h>

#define CONSTRUCTOR_BODY_OF_STRUCT(type)
```

Expands to the code required by a constructor of the form **type** * type_new (...) to allocate objects of type **type**, filling them with zeroes. If the allocation was performed successfully, a pointer to the allocated object is returned. If there was no memory left to allocate the object, NULL is returned instead.

The purpose of this macro is to reduce the lines of code of constructors of simple objects without fields requiring special initializations, such as strings, mutexes and other allocatable structures. In that case, CONSTRUCT_STRUCT should be used instead.

```
#include <util.h>

#define MANDATORY(int action)
```

Ensures the action **action** is performed correctly. **action** can be any expression that evaluates to non-zero on success and zero on failure (boolean sense). If **action** fails, FAIL is called halting the microkernel.

```
#include <util.h>

#define LIKELY_TO_FAIL(int action)
#define LIKELY_TO_SUCCESS(int action)

#define UNLIKELY_TO_FAIL(int action)
#define UNLIKELY_TO_SUCCESS(int action)

#define PTR_LIKELY_TO_FAIL(void *action)
#define PTR_LIKELY_TO_SUCCESS(void *action)

#define PTR_UNLIKELY_TO_FAIL(void *action)
#define PTR_UNLIKELY_TO_SUCCESS(void *action)
```

Prefetch macros used in conditional sentences to help compiler branch prediction. Interrupts and other real-time code should rely in these macros when determining the status of any action. Failure and success is evaluated according to the data type of the evaluation of **action**. In the case of integer error codes, success and failure is determined by SUCCESS(**action**) and FAILURE(**action**) respectively. If **action** evaluates to a pointer, success and failure is determined by SUCCESS_PTR(**action**) and FAILED_PTR(**action**) respectively.

```
#include <util.h>

#define RETURN_ON_FAILURE(action)
#define RETURN_ON_PTR_FAILURE(action)

#define PTR_RETURN_ON_FAILURE(action)
#define PTR_RETURN_ON_PTR_FAILURE(action)
```

Return KERNEL_ERROR_VALUE (RETURN_) or KERNEL_INVALID_POINTER (PTR_RETURN_) if the **action** has failed as an integer error code (_ON_FAILURE) or as a pointer (_ON_PTR_FAILURE). Success and failure is determined via SUCCESS/FAILURE and SUCCESS_PTR/FAILED_PTR macros. This macro uses branch prediction.

```
#include <util.h>

#define IN_BOUNDS(int idx, int size)
```

Evaluates to 1 if **idx** is in the interval [0, size), and 0 otherwise. Intended for their use with array indices whose size is known. This macro uses branch prediction.

```
#include <debug.h>

#define debug(const char *fmt, ...)
#define warning(const char *fmt, ...)
#define error(const char *fmt, ...)
```

Write formatted debug messages to the boot console according to their severity level (debug, warning or error). This macros expand to empty lines if NDEBUG is defined before the inclusion of debug.h.

```
#include <debug.h>

#define ASSERT(condition)
```

Checks whether **condition** evaluates to non-zero, otherwise outputs an error message to the boot console, issues a bugcheck interrupt and halts the microkernel. The difference between MANDATORY and ASSERT is that ASSERT can be removed safely without altering the behavior of the code. ASSERT expands to an empty line if NDEBUG is defined before the inclusion of debug.h.

```
#include <debug.h>

#define DEBUG_FUNC(function_name)
#define DEBUG_VAR(variable_name)
```

Exports the symbol **variable_name** or **function_name** according to their type, making their names visible to a backtrace. These macros expect a symbol name only, not the whole declaration / prototype.