Software

INTEL'S CONTRIBUTION FOR JERRYSCRIPT IN 2017

Zidong Jiang

zidong.jiang@intel.com

Main Patches From Intel

- es2015 features
  - TypedArray
  - Promise
- native pointer & type info
- jerry extensions
  - arguments validation
  - auto release value
  - module
- multiple instance

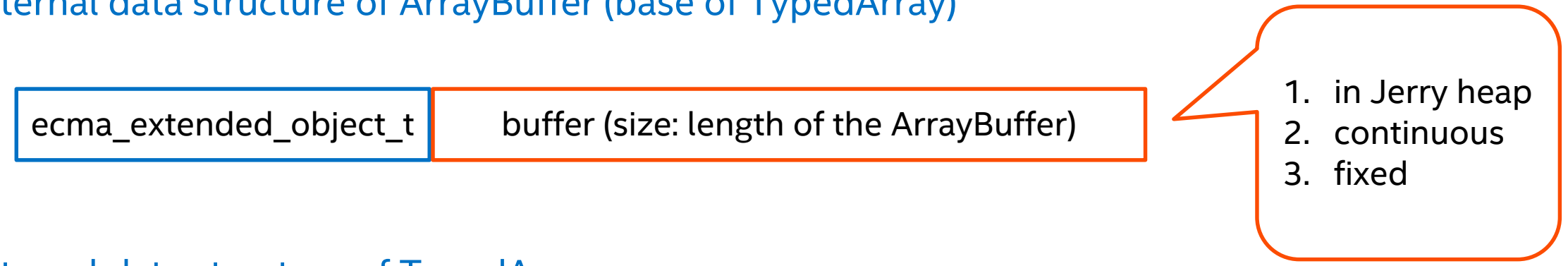
ES2015 Features

- Motivation: to implement W3C sensor/device related API
- Status:
  - 9 kinds of TypedArray (not a full impl.)
  - Promise (full + C API)

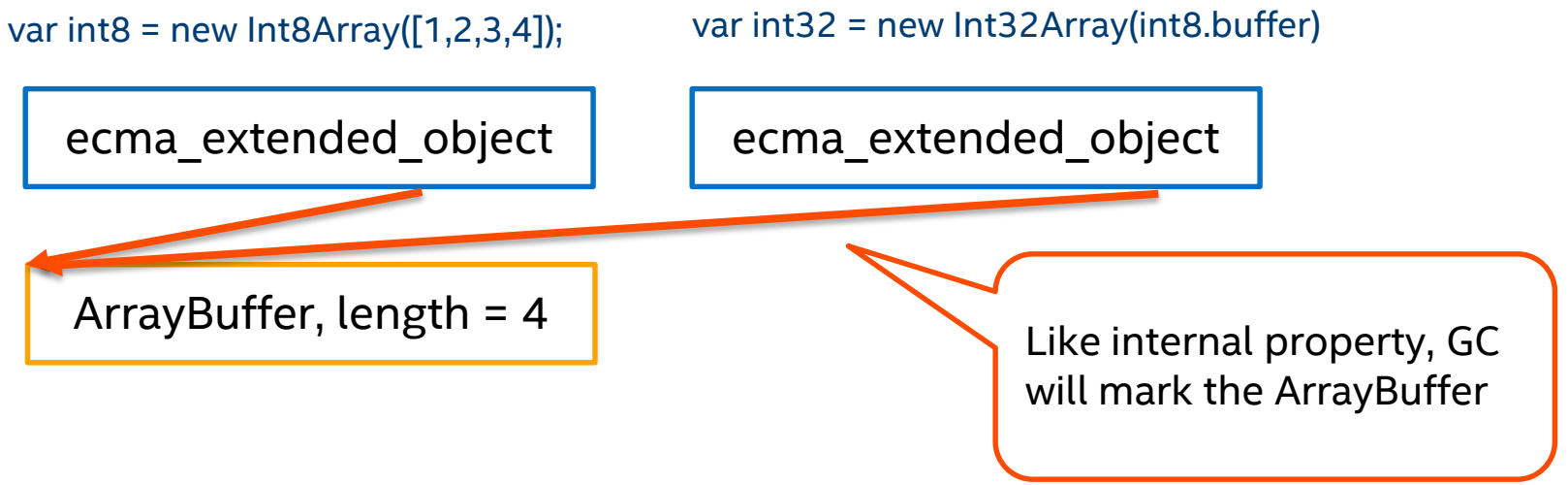
- add a es2015-subset profile
  - python tools/build.py --profile=es2015-subset
- or select features manually
  - CONFIG_DISABLE_ES2015_PROMISE_BUILTIN
  - CONFIG_DISABLE_ES2015_TYPEDARRAY_BUILTIN

Internal Details of TypedArray

- Internal data structure of ArrayBuffer (base of TypedArray)



- Internal data structure of TypedArray



Internal Details of Promise

```
var p = new Promise(function(d,r){
    d(1);
})
p.then(print); // It will not execute the "print" immediately,
               // but enqueues a job to the promise queue,
               // and run after current job
print(2)
```

jerry_init()

jerry_parse()

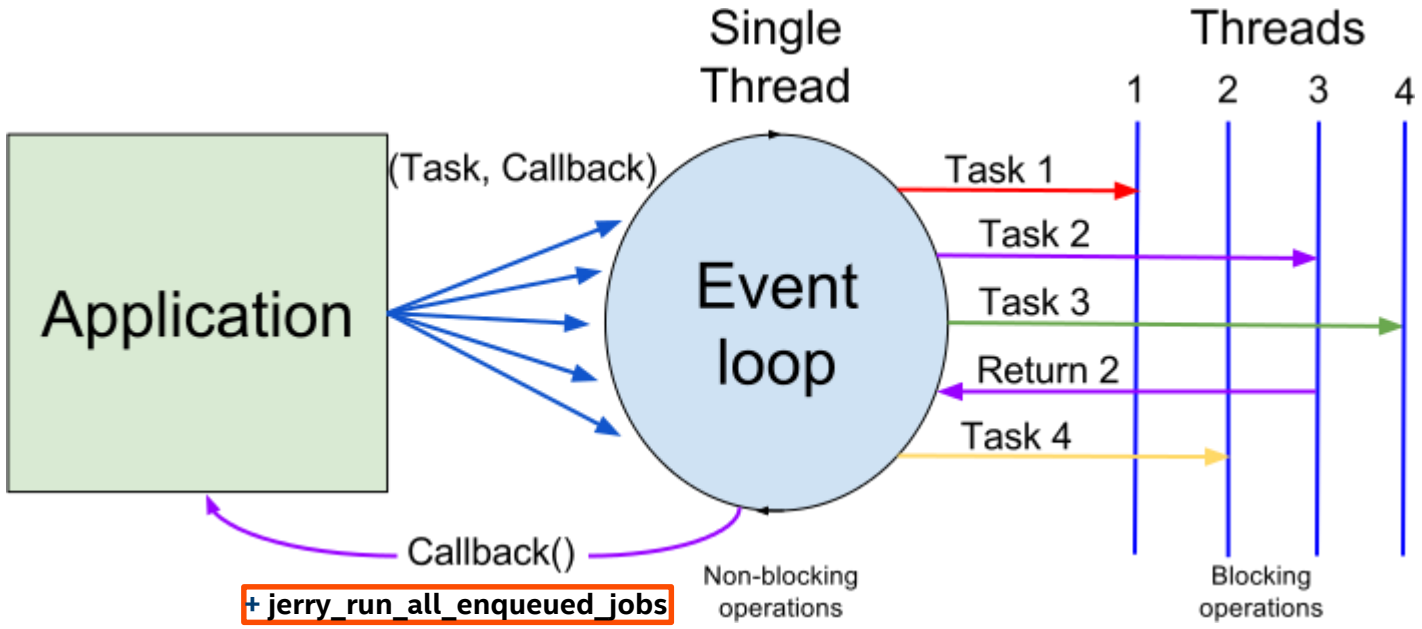
jerry_run()

// '2' is printed

jerry_run_all_enqueued_jobs()

(Akos #1804)

// '1' is printed



Internal Details of Promise

```
var p = new Promise(function(d,r){
    d(1);
})
p.then(print); // It will not execut
               // but enqueues a jo
               // and run after cur
print(2)
```

```
bool more;
do {
  more = uv_run(iotjs_environment_loop(env), UV_RUN_ONCE);
  more |= iotjs_process_next_tick();
  if (more == false) {
    more = uv_loop_alive(iotjs_environment_loop(env));
  }
  jerry_value_t ret_val = jerry_run_all_enqueued_jobs();
  if (jerry_value_has_error_flag(ret_val)) {
    DLOG("jerry_run_all_enqueued_jobs() failed");
  }
} while (more && !iotjs_environment_is_exiting(env));
```

jerry_init()

jerry_parse()

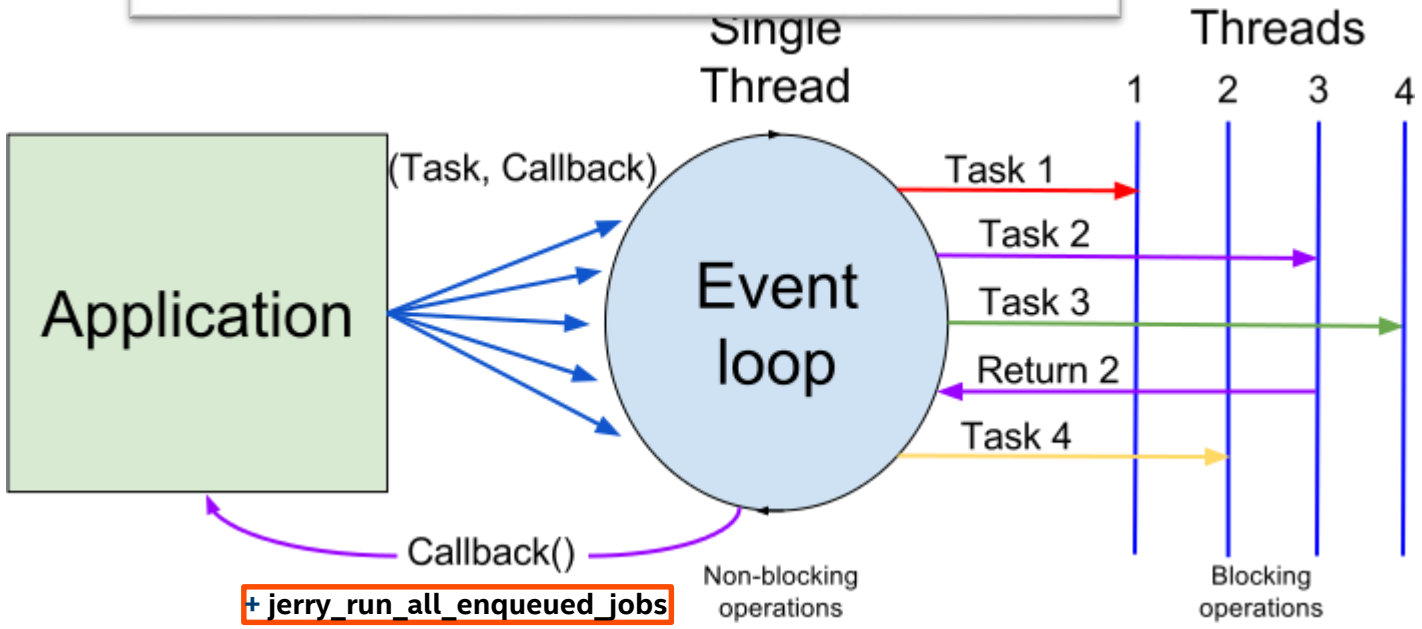
jerry_run()

// '2' is printed

jerry_run_all_enqueued_jobs()

(Akos #1804)

// '1' is printed



Type Information For Native Pointer

```
var d = new DigitalIO(1);
// native_digital_t *d_native = …;
// jerry_set_object_native_handle (d_js, &d_native, freecb);

var a = new AnalogIO(2);
// native_analog_t *a_native = …;
// jerry_set_object_native_handle (a_js, &a_native, freecb);

a.analogRead.call(d)
// `this` in analogRead is d_js, but the expected native pointer is a_native
// native_analog_t *a_native;
// jerry_get_object_native_handle (d_js, &a_native);
// The Pointer inside 'd' is native_digital, but treated as native_analog
// Point type cast, NOT SAFE!
```

Solution:

1. When set native handle, save the native's type info
2. When get native handle, load the type info and check

How to represent the type info?

Before:

- Jerry object stores a pointer to the freecb

Now:

- Jerry object store a pointer to a `jerry_object_native_info_t` struct
  - its member is freecb
  - the pointer value (the address itself) is the “type”

Recommendations:
define `jerry_object_native_info_t` instance for each native type, to make sure the address is unique and unchanged

see help doc for details

```
/**
 * Type information of a native pointer.
 */
typedef struct
{
  jerry_object_native_free_callback_t free_cb; /**< the free callback of the native pointer */
} jerry_object_native_info_t;
```

```
bool jerry_get_object_native_pointer (const jerry_value_t obj_val,
                                      void **out_native_pointer_p,
                                      const jerry_object_native_info_t **out_pointer_info_p);
void jerry_set_object_native_pointer (const jerry_value_t obj_val,
                                      void *native_pointer_p,
                                      const jerry_object_native_info_t *native_info_p);
```

Example

```
1  static const jerry_object_native_info_t native_type_analog_io =
2  {
3    .free_cb = native_free_cb_for_analog_io,
4  }
5
6  // in analogio creation
7  {
8    ...
9    native_analog_t *a_native = get_analog_io();
10   jerry_set_object_native_pointer(a_js, &a_native, &native_type_analog_io);
11   ...
12 }
13
14
15 // in external funtion for analog read
16 {
17   ...
18   native_analog_t *native_p;
19   const jerry_object_native_info_t *type_p;
20   bool has_p = jerry_get_object_native_pointer (this_val, &native_p, &type_p);
21
22   if (type_p == &native_type_analog_io)
23   {
24     // TYPE MATCH!
25     int val = native_analog_read (native_p);
26   }
27   else
28   {
29     // TYPE NOT MATCH!!!!
30     // this_val is not a JS obj for analog
31   }
32 }
```

define type info for native_analog

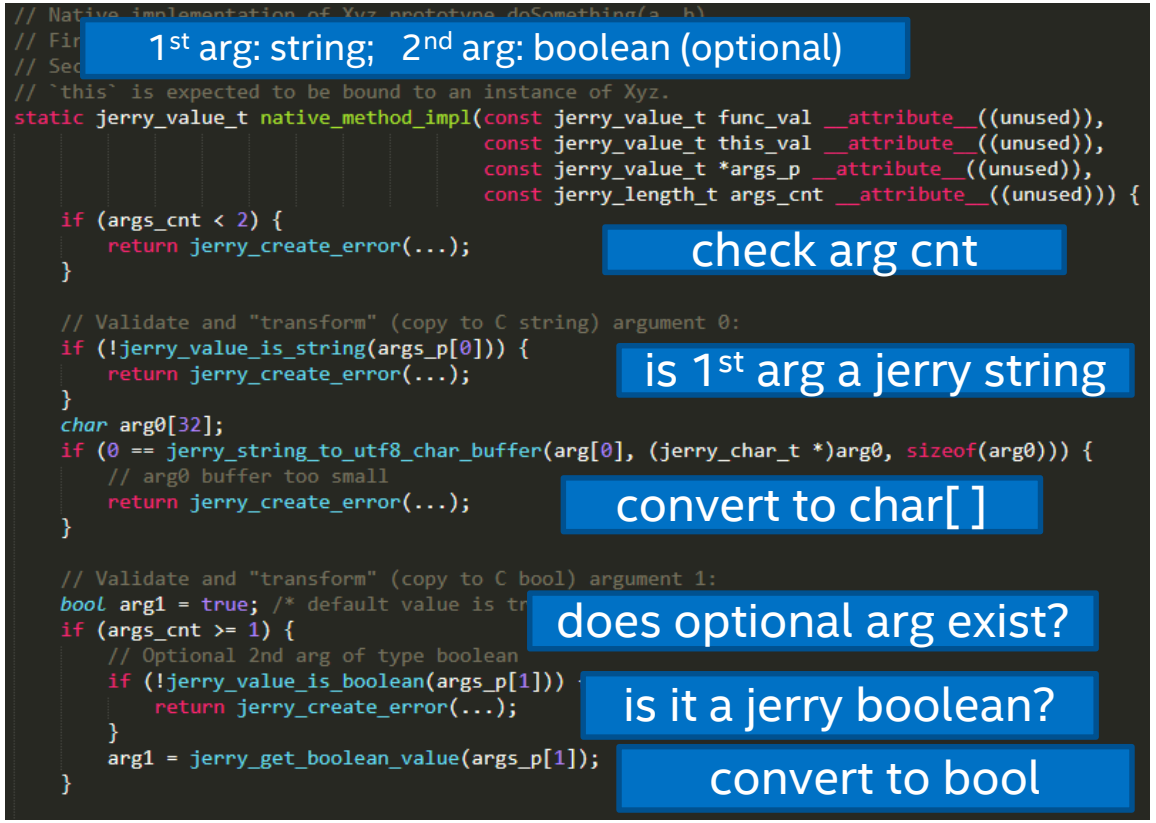
save type info to analog js obj

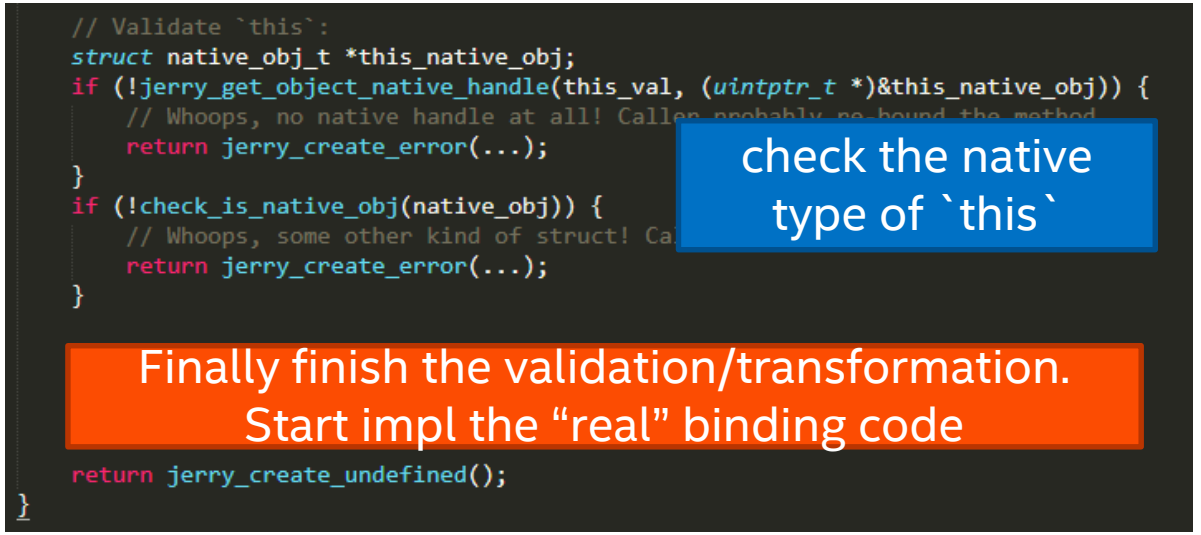
check whether the type info is expected

Is it annoying to write such code?

Argument Validation in External Functions

Problem: lots of trivial code to check/transform the arguments in external function for binding.





- Almost all project are using helper function to condense the validation code, but not covering all cases and not compatible.
- Avoid “rebuilding the wheel”

- **Add jerry-ext: a collection of helpers/tools in binding**
  - **API prefix “jerryx_”**

Example of argument validation in jerry-ext

- jerryx_arg_uint8
- jerryx_arg_uint16
- jerryx_arg_uint32
- jerryx_arg_int8
- jerryx_arg_int16
- jerryx_arg_int32
- jerryx_arg_number
- jerryx_arg_boolean
- jerryx_arg_string
- jerryx_arg_function
- jerryx_arg_native_pointer
- jerryx_arg_object_properties
- jerryx_arg_custom

declare what we want

start validation and transform

check result

business code

```
static const jerry_native_handle_info_t native_obj_info = ...;

// Native implementation of Xyz.prototype.doSomething(a, b)
// First arg is expected to be a string.
// Second (optional) arg is expected to be a boolean.
// `this` is expected to be bound to an instance of Xyz.
static jerry_value_t native_method_impl(const jerry_value_t func_val,
                                        const jerry_value_t this_val,
                                        const jerry_value_t *args_p,
                                        const jerry_length_t args_cnt) {
  char arg0[32];
  bool arg1 = true; /* default value is true */
  struct native_obj_t *this_native_obj;

  const jerry_arg_t mapping[] = {
    // First element in the array is the mapping for `this`.
    jerryx_arg_native_pointer(&this_native_obj, &native_obj_info, JERRYX_ARG_REQUIRED),
    // Further elements map to args_p[0], args_p[1], etc.
    jerryx_arg_string(arg0, sizeof(arg0), JERRYX_ARG_COERCE, JERRYX_ARG_REQUIRED),
    jerryx_arg_boolean(&arg1, JERRYX_ARG_NO_COERCE, JERRYX_ARG_OPTIONAL),
  };

  jerry_value_t mapping_result = jerryx_arg_transform_args(
      args, args_cnt, mapping, ARRAY_SIZE(mapping));

  if (jerry_value_has_error_flag(mapping_result))
  {
    return mapping_result;
  }

  jerry_release_value(mapping_result)

  // yay, arguments validated (that was easy!)
  // Now, let's use this_native_obj, arg0 and arg1 to do something interesting!
  // ... method implementation here ...

  return jerry_create_undefined();
}
```

Code Size Comparison

The binary size of the external function.

```c
static jerry_value_t
manual2( __attribute__((unused)) const jerry_value_t func_obj_val,
         __attribute__((unused)) const jerry_value_t this_val,
         const jerry_value_t *   args,
         const jerry_length_t    args_cnt)
{
  __attribute__((unused)) double num1;
  __attribute__((unused)) double num2;

  if (args_cnt != 2
    || !jerry_value_is_number (args[0])
    || !jerry_value_is_number (args[1])) {
    return jerry_create_error (JERRY_ERROR_TYPE, (const jerry_char_t*)"");
  }

  num1 = jerry_get_number_value (args[0]);
  num2 = jerry_get_number_value (args[1]);

  return jerry_create_undefined ();
}
```

```c
static jerry_value_t
auto2( __attribute__((unused)) const jerry_value_t func_obj_val,
       __attribute__((unused)) const jerry_value_t this_val,
       const jerry_value_t *   args,
       const jerry_length_t    args_cnt)
{

  double num1;
  double num2;

  const jerryx_arg_t mapping[] = {
    jerryx_arg_number(&num1, JERRYX_ARG_NO_COERCE, JERRYX_ARG_REQUIRED),
    jerryx_arg_number(&num2, JERRYX_ARG_NO_COERCE, JERRYX_ARG_REQUIRED),
  };

  jerry_value_t rv = jerryx_arg_transform_args(args, args_cnt, mapping, 2);

  if(jerry_value_has_error_flag (rv))
  {
    return rv;
  }

  jerry_release_value (rv);
  return jerry_create_undefined ();
}
```

Arguments	x86-32 (auto - manual)	ARMv7 (auto - manual)
1 arg	+10 byte	+17 byte
2 args	-20 byte	-7 byte
3 args	-50 byte	-22 byte

AutoRelease Value and Module (in jerry-ext)

AutoRelease value: reduce lots of “jerry_release_value()” in binding code.

- Refer to the similar macro in Zephyr.js
- Depends on compiler, __attribute__ ((__cleanup__(...)))

```
{
    JERRYX_AR_VALUE_T global = jerry_get_global_object();
    …
    // no need to call jerry_release_value(global);
}
// When the `global` go out of scope, automatically call jerry_release_value
```

Module: load modules (use it to load native module or “require” js module)

- See Gabriel’s sharing for details.

Support Multiple Instance

```
typedef struct
{
  jmem_heap_free_t first; /**< first node in free region list */
  uint8_t area[JMEM_HEAP_AREA_SIZE]; /**< heap area */
} jmem_heap_t;

/**
 * Global heap.
 */
extern jmem_heap_t jerry_global_heap;
```

```
/**
 * Global context.
 */
extern jerry_context_t jerry_global_context;
```

```
/**
 * Provides a reference to a field in the current context.
 */
#define JERRY_CONTEXT(field) (jerry_global_context.field)
```

Problem:

1. Can’t configurate heap size during runtime.
2. One global context: can’t run multi JS apps in multi-thread

Ideas:

1. prepare jerry-heap in runtime
2. no global heap/context, use current heap/context instead

Support Multiple Instance: Solution

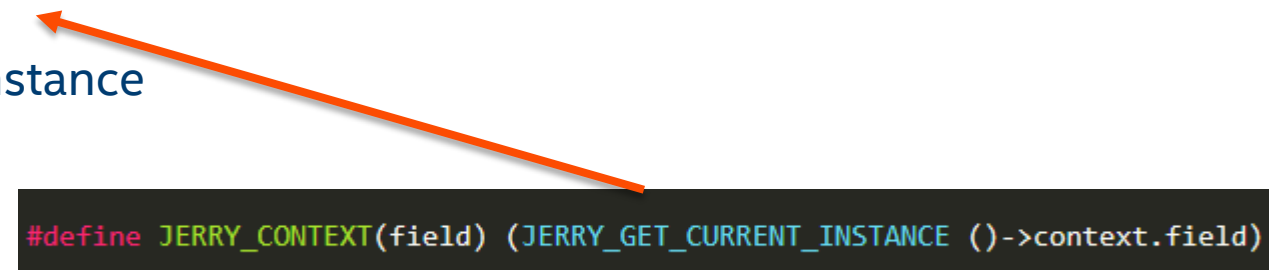
Define macro JERRY_ENABLE_EXTERNAL_CONTEXT to enable the feature.

JERRY API: jerry_create_instance (heap_size, ...)

- create jerry_instance (context + heap + lcache)

PORT API: jerry_port_get_current_instance

- Jerry will call them to get the current instance



- Multi-thread case:
  - prepare instances for each thread
  - store them in thread local storage.
  - `jerry_port_get_current_instance` get the current instance via TLS API.

Q&A