Tizen/Artik IoT Lecture Chapter 4. JerryScript ECMA Internal and Memory Management

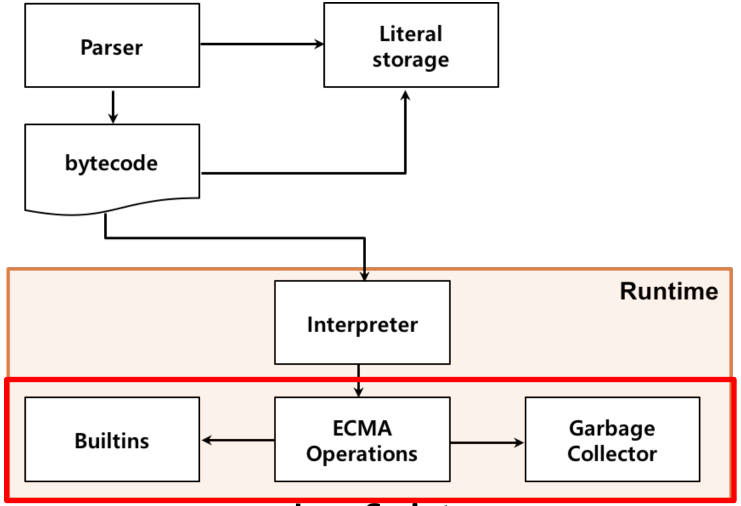
Sungkyunkwan University

JerryScript and ECMAScript Overview

- **ECMAScript**
  - ECMAScript: Script-language specification standardized by ECMA International in ECMA-262
  - JerryScript is fully compatible with ECMA-262 edition 5.1

❖ ECMA-262 provides definitions of operation and data representation

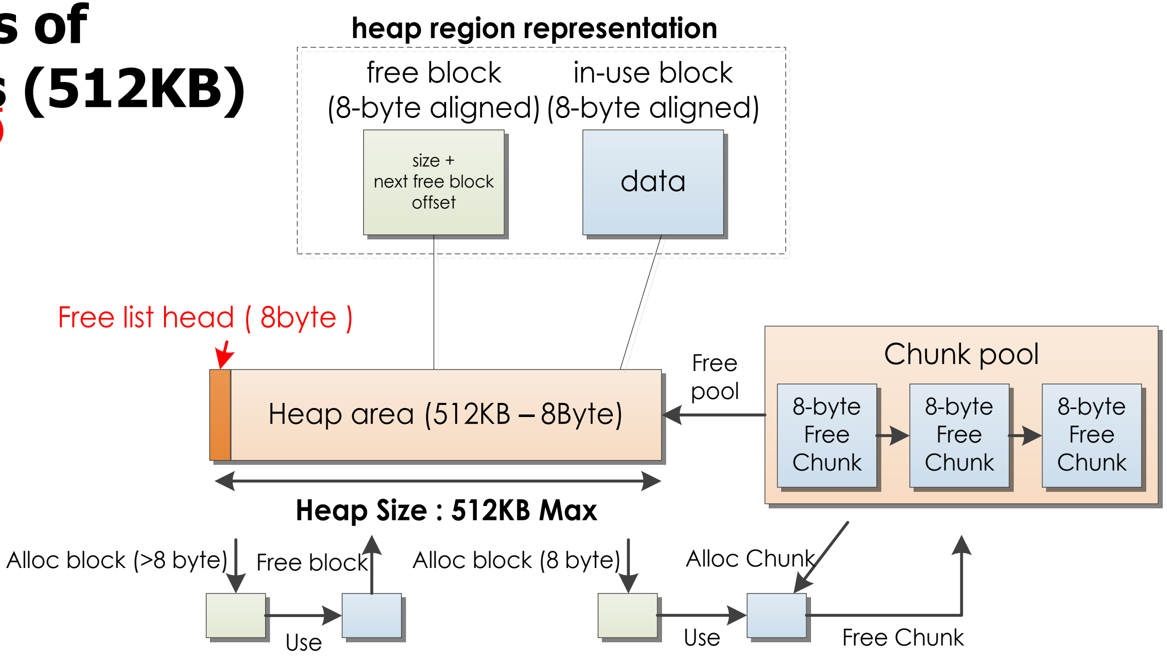
❖ Every JavaScript engine satisfies ECMA requirements in their own way



<JerryScript>

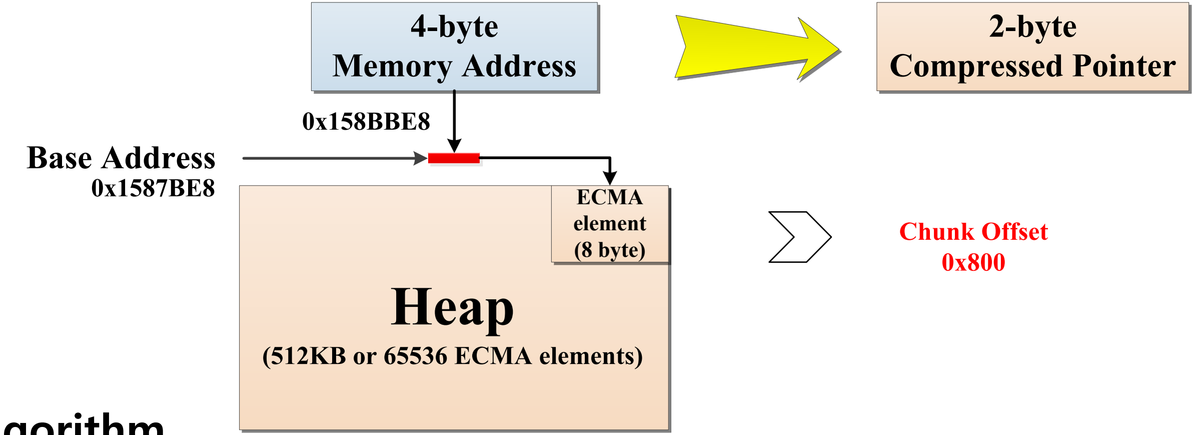
Heap Memory Architecture

- **Heap memory consists of maximum 64K chunks (512KB)** MAX(16bit)
- **8 byte alloc & free**
  - Alloc from pool
    → Alloc from heap
  - Free to pool
    → Free to heap
- **> 8 byte alloc & free**
  - Alloc from heap
  - Free to heap



ECMA Compressed Pointer (Compressing)

- **When pointer operations occur, compressed pointer will be decompressed or compressed**

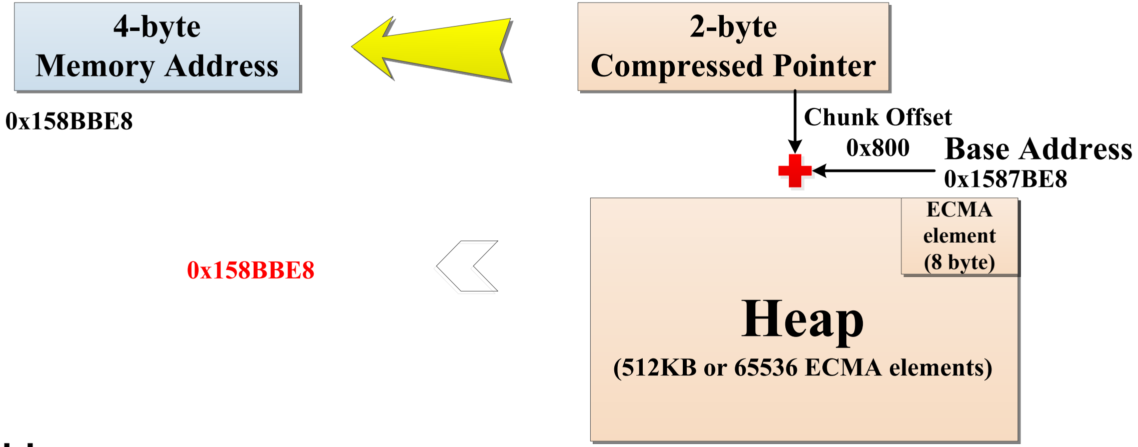


Compression Algorithm

1. Make sure the decompressed address is 8bytes aligned 0x158BBE8 % 8=0
2. Decompressed Address -= heap Start Address 0x158BBE8 - 0x1587BE8 = 0x4000
3. Decompressed Address >>= 3 0x4000 >> 3 = 0x800

8Bytes Alignment

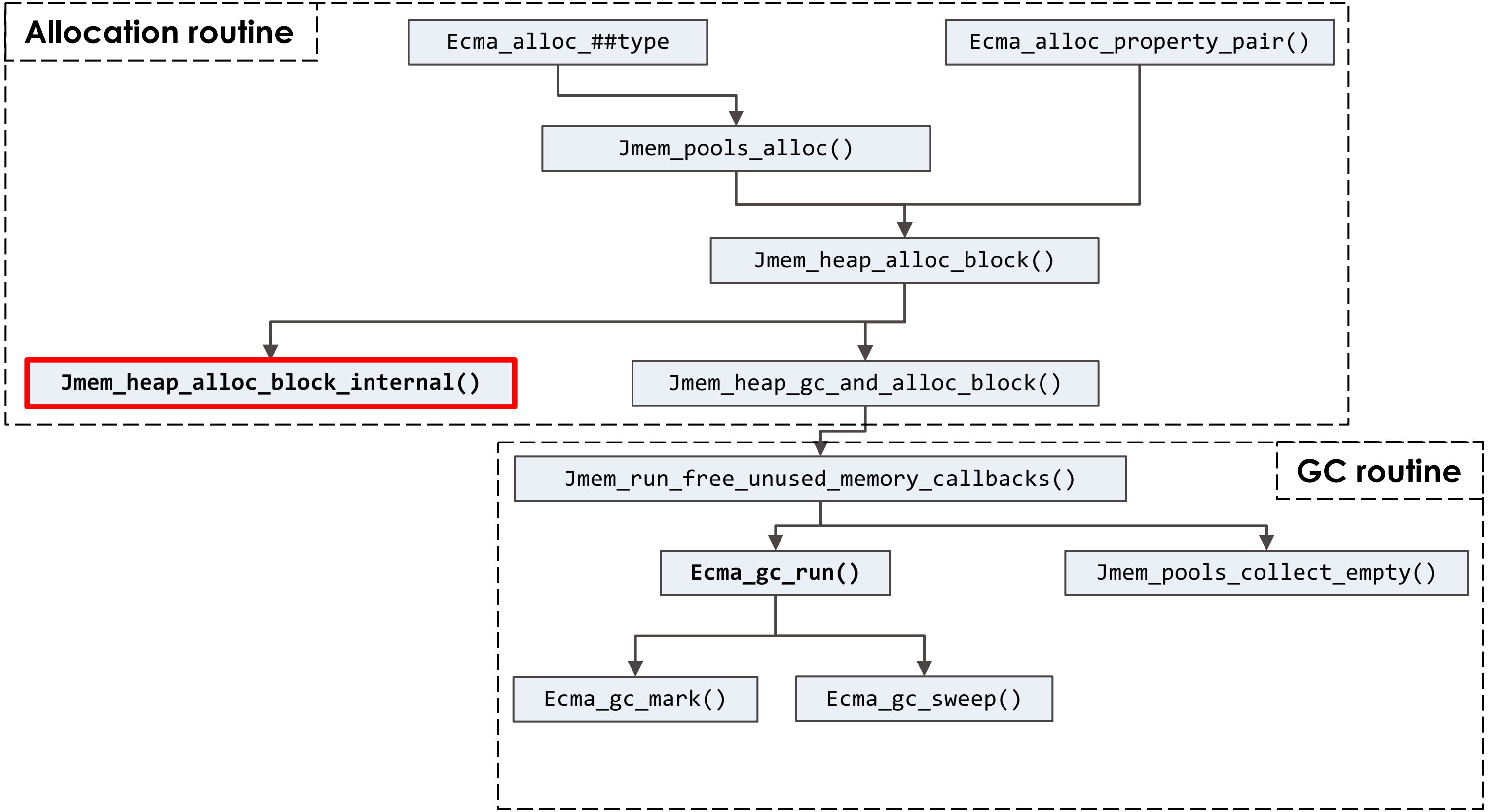
ECMA Compressed Pointer (Decompressing)



decompressing Address

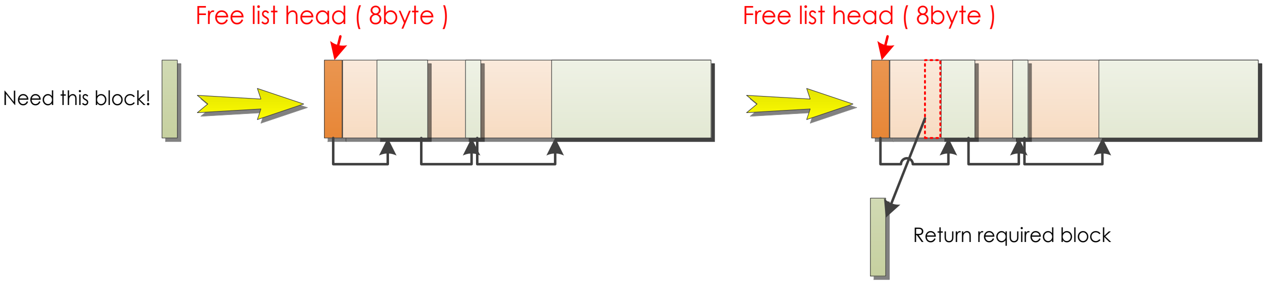
1. Make sure the compressed address is 8bytes aligned 0x800 % 8 = 0
2. Compressed address <<= 3 0x800 << 3 = 0x4000
3. Compressed address += heapstart 0x4000 + 0x1587BE8 = 0x158BBE8

Function Call Routine


Allocation routine
Ecma_alloc_##type
Ecma_alloc_property_pair()
Jmem_pools_alloc()
Jmem_heap_alloc_block()
Jmem_heap_alloc_block_internal()
Jmem_heap_gc_and_alloc_block()
GC routine
Jmem_run_free_unused_memory_callbacks()
Ecma_gc_run()
Jmem_pools_collect_empty()
Ecma_gc_mark()
Ecma_gc_sweep()

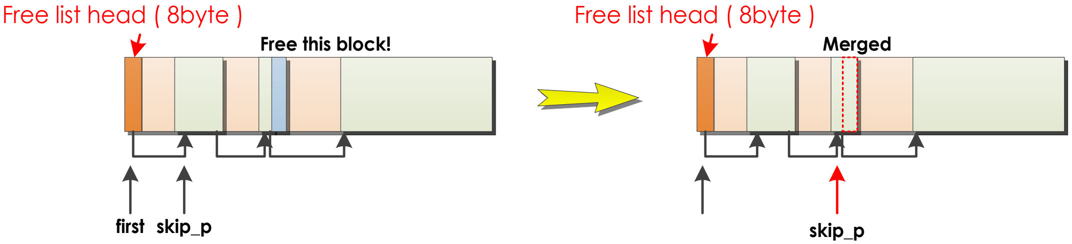
Jmem_heap_alloc_block_internal

- **Fast Path for 8 byte block (A chunk)**
  - Every free region is guaranteed to be sufficient for 8 byte chunk
    → Just allocate from first free region (No need to check the region size)
- **Slow Path**
  - Check each free region if it is sufficient for required memory size
    → Required Memory size (8-byte aligned) <= Free region size
- **If a free region first-fitted with required size is found, split required-sized block from the region.**



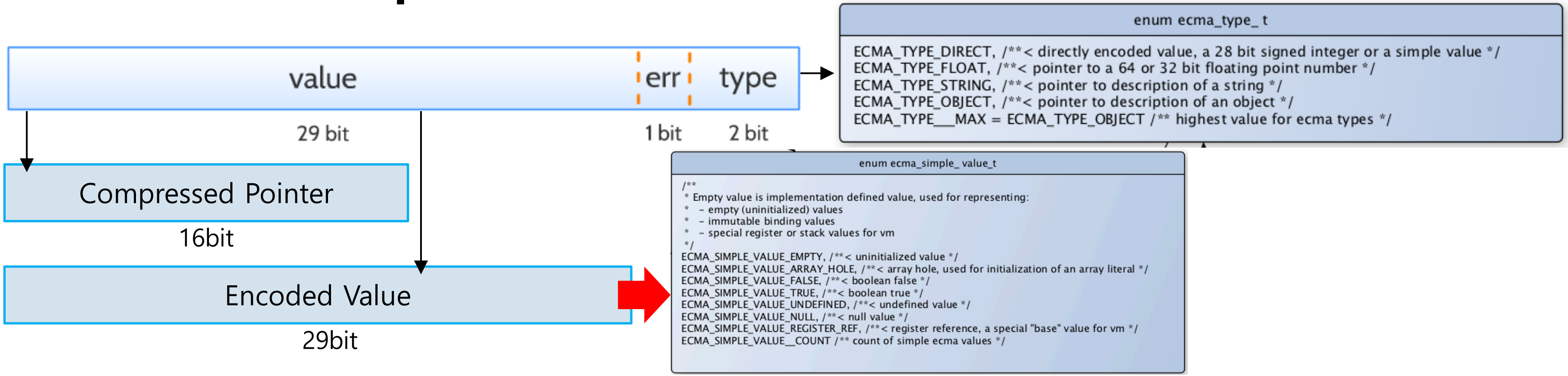
Jmem_heap_free_block

- **When a heap block is freed, each free region in heap should be checked if it is able to be merged.**
  - Check the neighbor free region of the block to be freed (Previous region and Next region)
- **Lookup the neighbor free region**
  - *"Next block of previously freed block would be freed successively"*
  - Linear Search **from first** OR **from previously freed region**



ECMA Representation

- **ECMA component of the engine is responsible for the following notions**
  - Data representation { Object, Number, String, Simple }
  - Runtime representation { Hashing, Lcache, Property Lookup, etc … }
  - Garbage collection (GC)
- **ECMA Value Representation**



enum ecma_type_ t

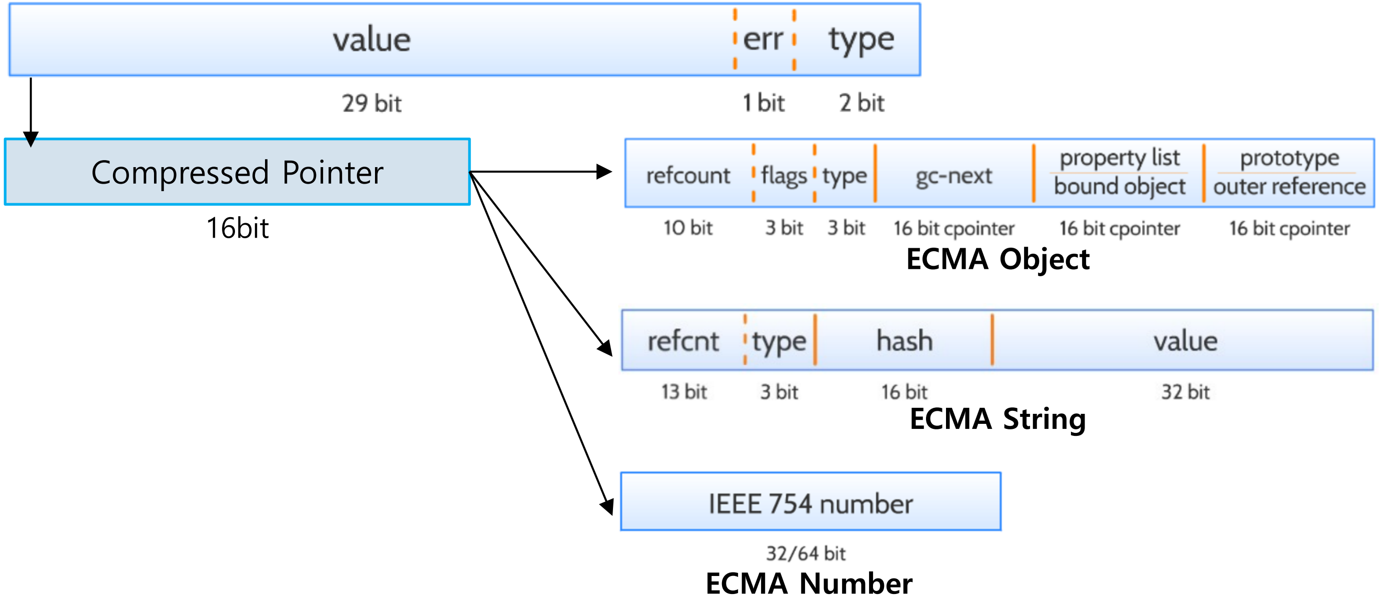
```
ECMA_TYPE_DIRECT, /**< directly encoded value, a 28 bit signed integer or a simple value */
ECMA_TYPE_FLOAT, /**< pointer to a 64 or 32 bit floating point number */
ECMA_TYPE_STRING, /**< pointer to description of a string */
ECMA_TYPE_OBJECT, /**< pointer to description of an object */
ECMA_TYPE__MAX = ECMA_TYPE_OBJECT /** highest value for ecma types */
```

enum ecma_simple_ value_t

```
/**
 * Empty value is implementation defined value, used for representing:
 *   - empty (uninitialized) values
 *   - immutable binding values
 *   - special register or stack values for vm
 */
ECMA_SIMPLE_VALUE_EMPTY, /**< uninitialized value */
ECMA_SIMPLE_VALUE_ARRAY_HOLE, /**< array hole, used for initialization of an array literal */
ECMA_SIMPLE_VALUE_FALSE, /**< boolean false */
ECMA_SIMPLE_VALUE_TRUE, /**< boolean true */
ECMA_SIMPLE_VALUE_UNDEFINED, /**< undefined value */
ECMA_SIMPLE_VALUE_NULL, /**< null value */
ECMA_SIMPLE_VALUE_REGISTER_REF, /**< register reference, a special "base" value for vm */
ECMA_SIMPLE_VALUE__COUNT /** count of simple ecma values */
```

ECMA Value Structure

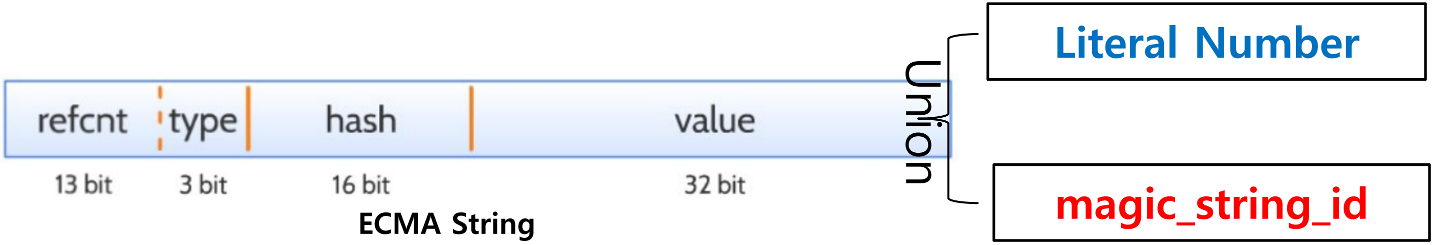
- **Mostly ECMA values contain the compressed pointer which points the object, number or string.**



String Representation

- **Statically assigned string data is stored in .rodata section**
- **Dynamically allocated string data is stored in heap**
  - The static string data which located in JavaScript file is stored in literal storage at parsing phase
  - The dynamic string data is allocated in jerry heap

- **Magic ID**
  - Reduce the memory overhead and computational overhead


refcnt | type | hash | value

13 bit | 3 bit | 16 bit | 32 bit

ECMA String

Union

Literal Number

magic_string_id

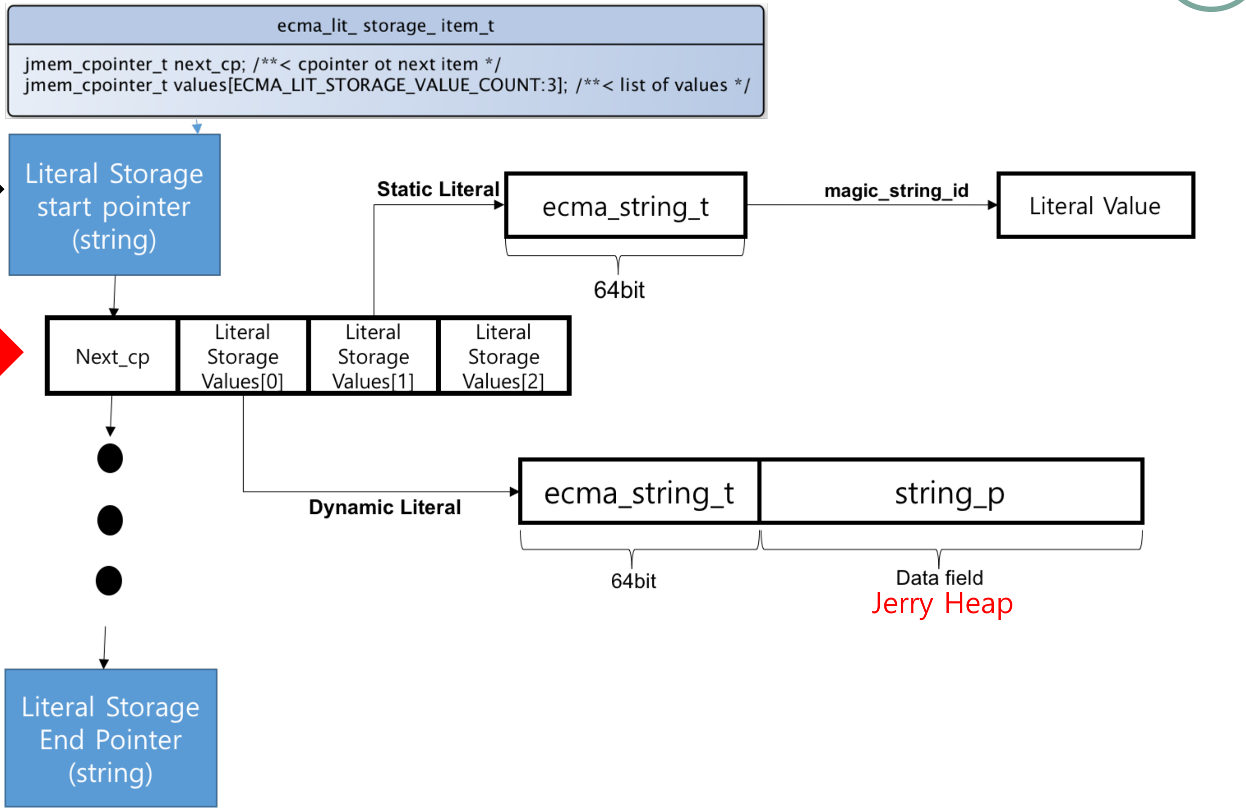
e.g var num = 295731;

e.g var str = 'eslab';

String Structure

Initialized parsing phase

Insert literal storage node



ecma_lit_ storage_ item_t

```
jmem_cpointer_t next_cp; /**< cpointer ot next item */
jmem_cpointer_t values[ECMA_LIT_STORAGE_VALUE_COUNT:3]; /**< list of values */
```

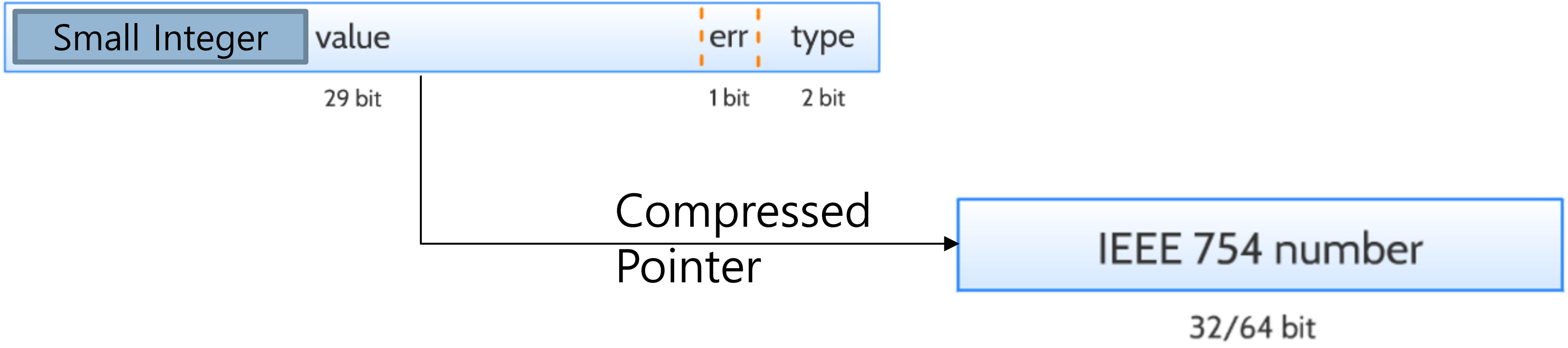
Literal Storage start pointer (string)
Static Literal
ecma_string_t
magic_string_id
Literal Value
64bit
Next_cp
Literal Storage Values[0]
Literal Storage Values[1]
Literal Storage Values[2]
Dynamic Literal
ecma_string_t
string_p
64bit
Data field
Jerry Heap
Literal Storage End Pointer (string)

Magic ID	String
85465	Seminar
4662	ESLAB

<Magic ID and String mapping>

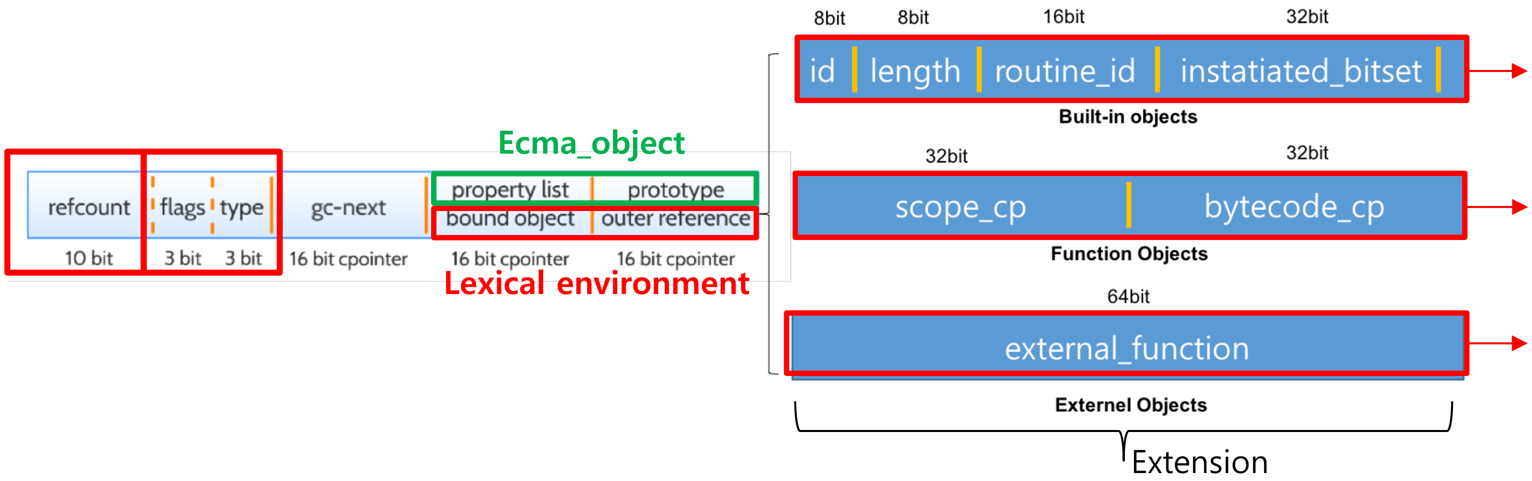
Number Representation

- **Jerryscript has two kinds of number representation**
  - 4-byte (Compact Profile)
  - 8-byte (Full Profile)
  - Small value in 29bit (ECMA_value section)



Object Representation

- **Object Types & Flags**
  - Types : 1. ecma_object, 2. ecma_lexical_environment
  - Flags : built-in Object, lexical environment, GC visited, Extensible
  - Reference-count : 10bit Reference Count for GC



Ecma_object

refcount | flags | type | gc-next | property list | prototype

bound object | outer reference

10 bit | 3 bit | 3 bit | 16 bit cpointer | 16 bit cpointer | 16 bit cpointer

Lexical environment

8bit | 8bit | 16bit | 32bit

id | length | routine_id | instatiated_bitset

Built-in objects

32bit | 32bit

scope_cp | bytecode_cp

Function Objects

64bit

external_function

Externel Objects

Extension

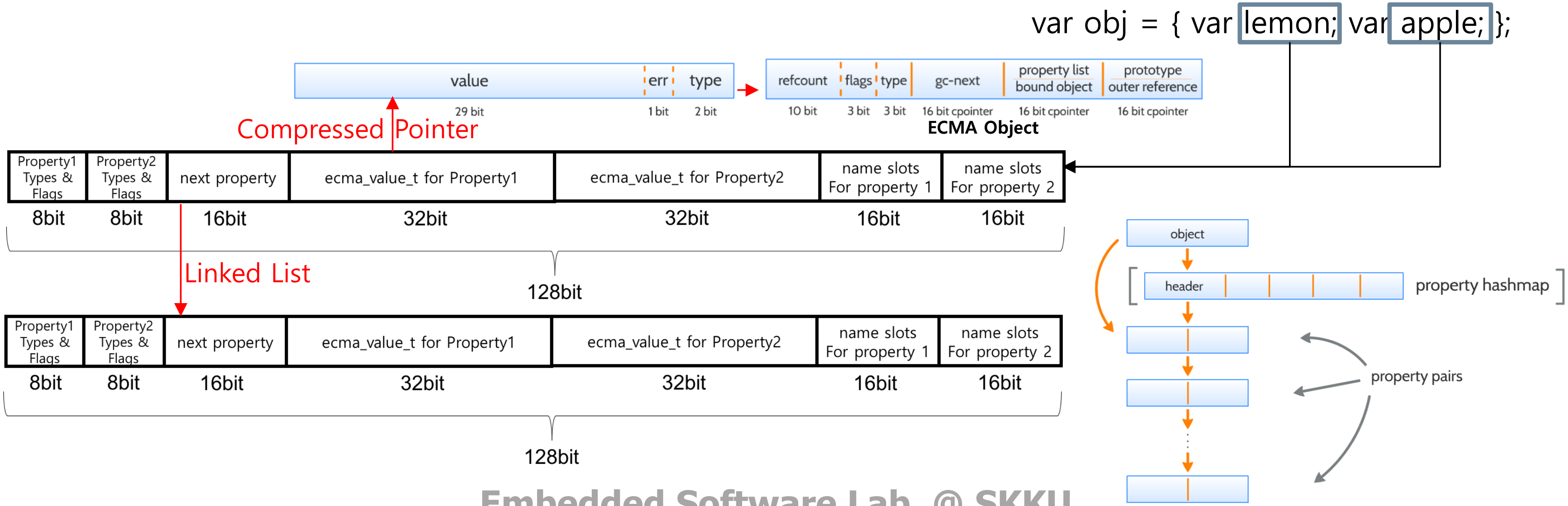
Built-in Objects
(Built-in Object)

Function Object
(Lexical Environment)

For Embedded API
Pointing to Native
Function

Object and Property Representation

- **Each property's metadata needs 72bit data structure**
  - In order to improve the CPU efficiency, jerry makes property pair (128bit)



var obj = { var lemon; var apple; };

value	err	type
29 bit	1 bit	2 bit

refcount	flags	type	gc-next	property list bound object	prototype outer reference
10 bit	3 bit	3 bit	16 bit cpointer	16 bit cpointer	16 bit cpointer

ECMA Object

Compressed Pointer

Property1 Types & Flags	Property2 Types & Flags	next property	ecma_value_t for Property1	ecma_value_t for Property2	name slots For property 1	name slots For property 2
8bit	8bit	16bit	32bit	32bit	16bit	16bit

128bit

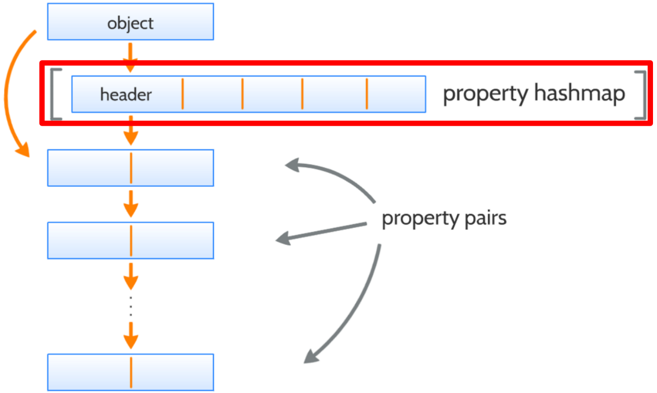
Linked List

Property1 Types & Flags	Property2 Types & Flags	next property	ecma_value_t for Property1	ecma_value_t for Property2	name slots For property 1	name slots For property 2
8bit	8bit	16bit	32bit	32bit	16bit	16bit

128bit

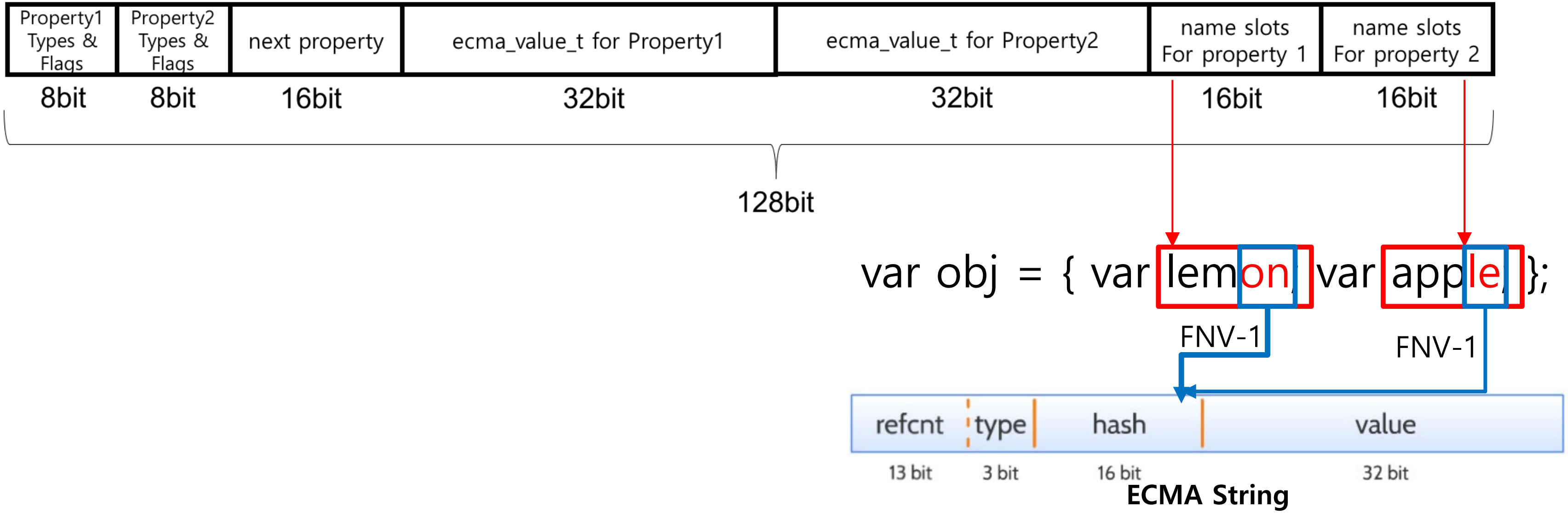
Property Hashmap cont’

- **Property hashmap is allocated in following condition.**
  - namedaccessor and nameddata property can be stored in hashmap
  - Property hashmap is dynamically allocated when the object has the number of properties more than 16 (Property Pair : 8)
  - The default # of properties is 32 entries
  - If the remainder of hash entries is under # of properties/8, jerry dynamically reallocates the bigger size of the hashmap
- **Property hashmap is allocated as a first entry in property linked-list**



Property Hashmap

- **The hash function is FNV-1 algorithm**
  - FNV hashes are designed to be fast while maintaining a low collision rate.
  - The end of two bytes (two character) of property name is used as a hash key



Property1 Types & Flags	Property2 Types & Flags	next property	ecma_value_t for Property1	ecma_value_t for Property2	name slots For property 1	name slots For property 2
8bit	8bit	16bit	32bit	32bit	16bit	16bit

128bit

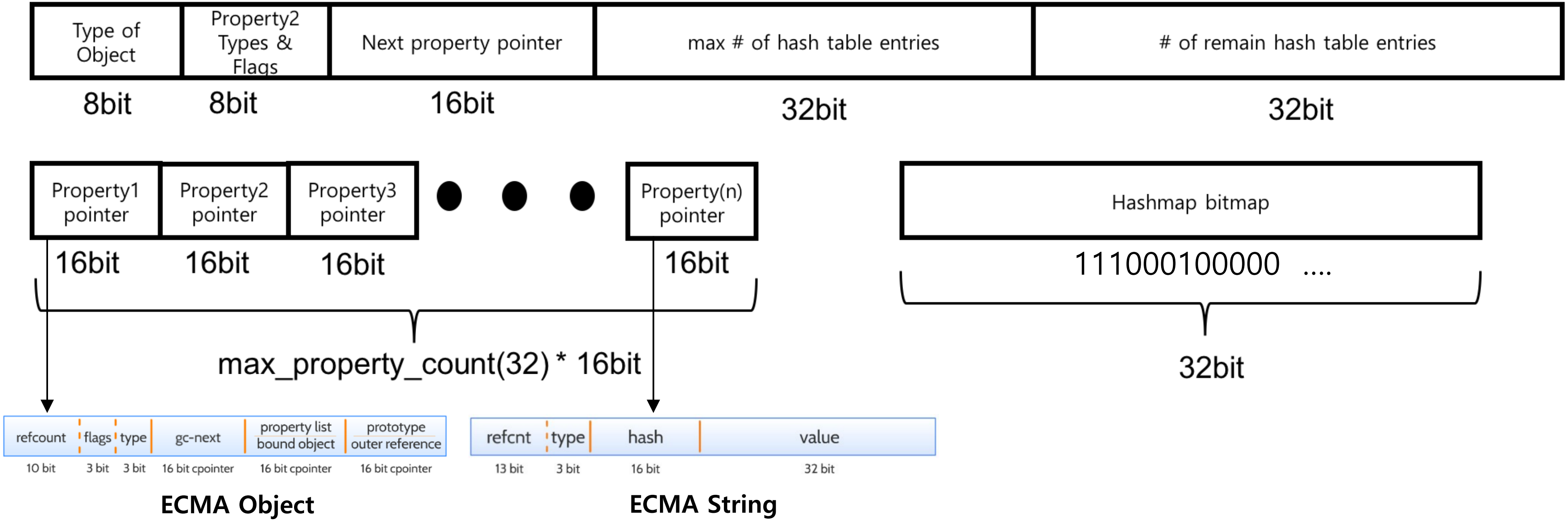
```
var obj = { var lemon, var apple, };
```

refcnt	type	hash	value
13 bit	3 bit	16 bit	32 bit

ECMA String

Property Hashmap

- ## Property Hashmap Structure



Type of Object	Property2 Types & Flags	Next property pointer	max # of hash table entries	# of remain hash table entries
8bit	8bit	16bit	32bit	32bit

Property1 pointer	Property2 pointer	Property3 pointer	● ● ●	Property(n) pointer
16bit	16bit	16bit		16bit

max_property_count(32) * 16bit

Hashmap bitmap
111000100000 ....

32bit

refcount	flags	type	gc-next	property list bound object	prototype outer reference
10 bit	3 bit	3 bit	16 bit cpointer	16 bit cpointer	16 bit cpointer

ECMA Object

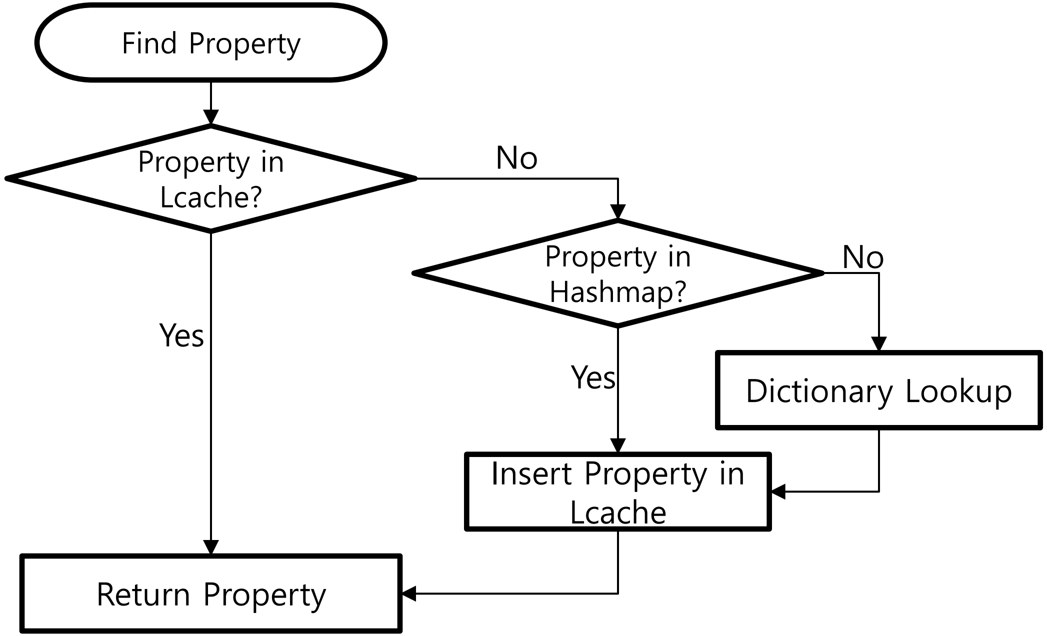
refcnt	type	hash	value
13 bit	3 bit	16 bit	32 bit

ECMA String

Lcache

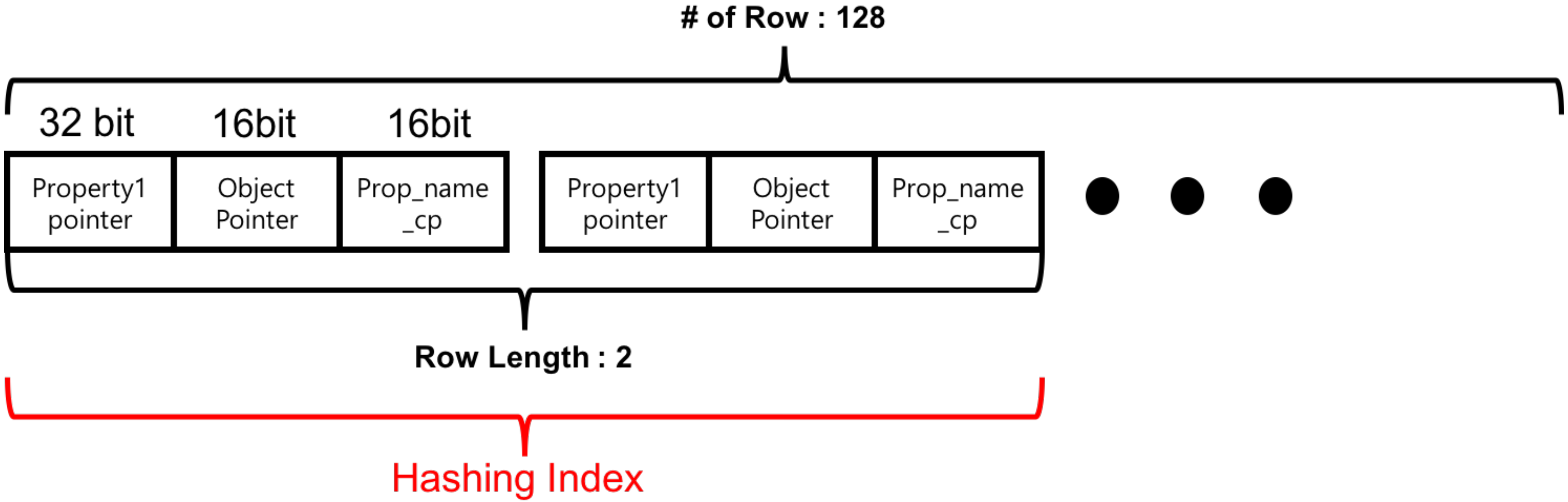
- **In order to find a property efficiently, jerry has global object-property mapping table (4KBytes)**
  - Property Find Algorithm

Even if hash allocation threshold (16 # of Properties) is too high, Lcache can improves performance



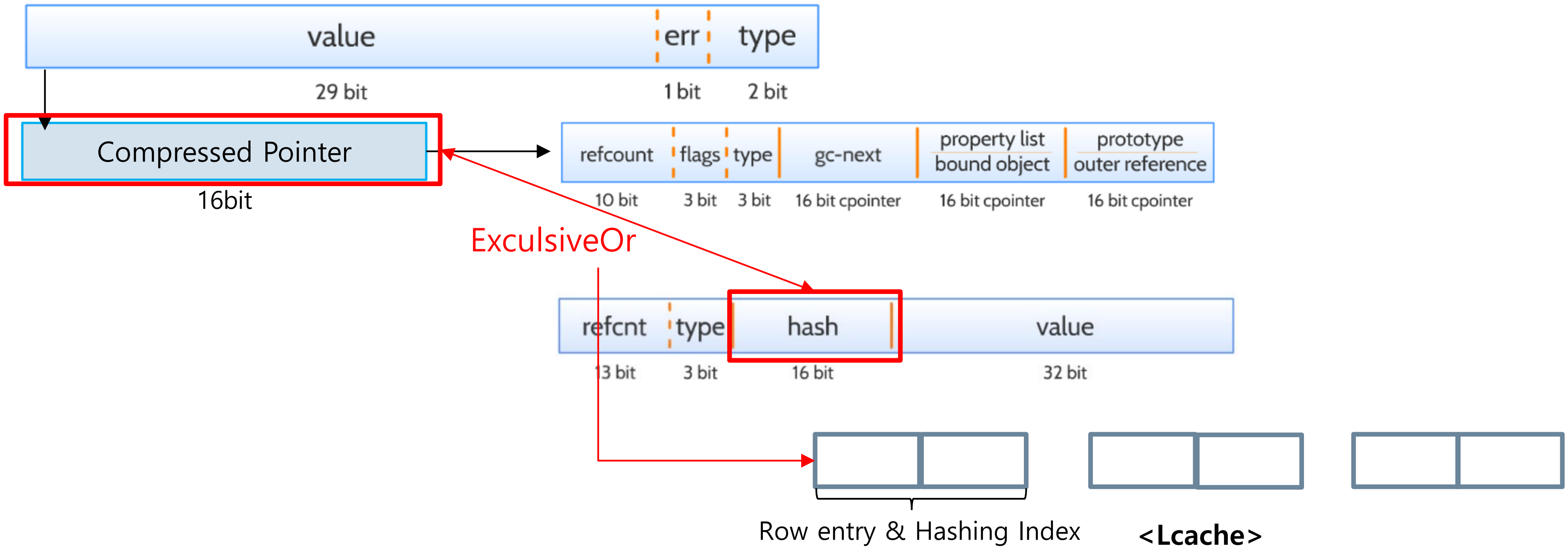
Lcache Structure

- **Lcache map size**
  - Row Length : 2, # of Row : 128
  - (32bit + 16bit + 16bit)*2 (Row Length)* 128 (# of Row) = 4KBytes
  - Lcache statically allocated in global section (.bss)



Lcache Operation

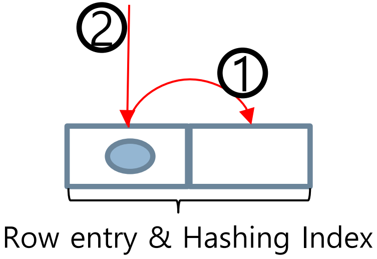
- **Indexing: Object address ^ Hash Value**



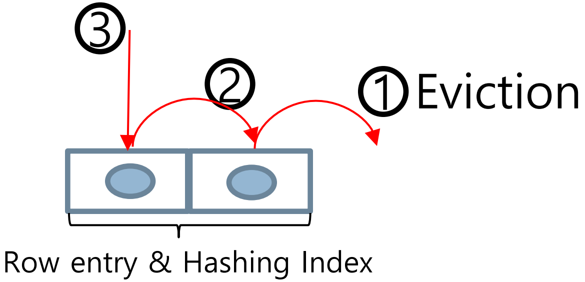
<Lcache>

Lcache Policy

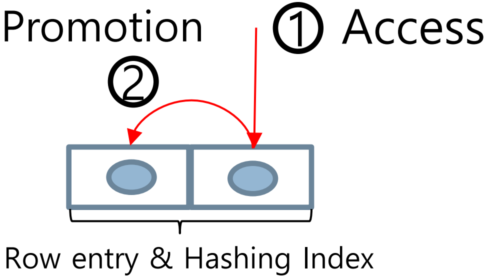
- **If collision occurs, use LRU eviction policy**
  - Same hash index property need to be inserted
  - lcache[index][1] = lcache[index][0]; move to old
  - lcache[index][0] = new property;
- **If the property located in old block is accessed, it is promoted to young block**



<only one block>



<block is full>



<promotion>

Memory Reclamation

- **Memory reclamation triggered**
  - At every 8KB allocation & At out of memory situation
- **Memory Reclamation (→ Garbage Collection)**

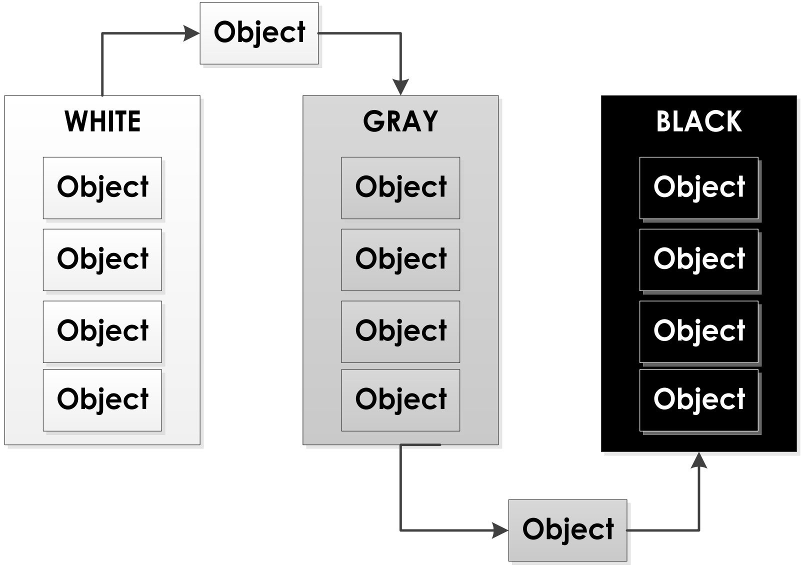
1. (GC with lower severity)
   - Invoked only if new objects are sufficiently allocated after last GC
     "Most of garbage collected is newly created post the previous GC cycle"
   - **Not free** property hashmap
2. Reclaim free chunks in the pool (pool is only for performance)

   `jmem_pools_collect_empty()`
3. GC with higher severity
   - **Free** property hashmap

Garbage Collection (Mark)

- **Tri-color marking**
  - **White (Unvisited)**
    : Not referenced by a live object or the reference not found yet.
  - **Gray (Visited)**
    : Referenced by some live object
  - **Black (Marked)**
    : Visited all the references of the object

Visit Ref.cnt > 0 object
Ref.cnt increased in object creation or object copy.
Ref.cnt decreased when object life-cycle ends



Visit the objects referenced by this object
(Insert them into GRAY)

Garbage Collection (Sweep)

- **After marking objects,**
  - In white
    - Garbage objects remain
  - In gray
    - No objects
  - In black
    - Live objects remain
- **Sweep all white objects and black objects transformed to white objects for next GC.**
  - Sweep: Free the object (jmem_heap_free_block invoked)