Nano-Kernel : A Bare Metal OS

## Part 7 - Standard Libraries

Since we are building are own OS kernel we do not have access to the user-space, OS provided set of libraries and tools, including things such as *printf*, *strcmp*, or even *memcpy*. In fact, we don’t even get access to the heap and *malloc*. Our goal in this section is to create naive implementations of these essential functions.

I use the term naive because we are going to use basic C function implementations. Professional kernels will have hand-optimized versions of many of these. Compare our version of *strncpy:*

|  |
| --- |
| char \* strncpy(char \* dest, const char \*src, int n) {  int i = 0;  while ((i < n) && (src[i] != 0)) {  dest[i] = src[i];  i++;  }  dest[i] = 0;  return dest;  } |

To the hand-written implementation of *strncpy* from Linux kernel’s “string\_32.c”:

|  |
| --- |
| #ifdef \_\_HAVE\_ARCH\_STRNCPY char \*strncpy(char \*dest, const char \*src, size\_t count) {  int d0, d1, d2, d3;  **asm** **volatile**("1:\tdecl %2\n\t"  "js 2f\n\t"  "lodsb\n\t"  "stosb\n\t"  "testb %%al,%%al\n\t"  "jne 1b\n\t"  "rep\n\t"  "stosb\n"  "2:"  : "=&S" (d0), "=&D" (d1), "=&c" (d2), "=&a" (d3)  : "0" (src), "1" (dest), "2" (count) : "memory");  **return** dest; } EXPORT\_SYMBOL(strncpy); #endif |

The eager-beavers of the world might choose to spend their time hand-rolling these core functions. In 2009, Randall Hyde wrote[[1]](#footnote-1):

Every programmer with a few years' experience or education has heard the phrase "premature optimization is the root of all evil." This famous quote by Sir Tony Hoare (popularized by Donald Knuth) has become a best practice among software engineers. Unfortunately, as with many ideas that grow to legendary status, the original meaning of this statement has been all but lost and today's software engineers apply this saying differently from its original intent. As computer systems increased in performance from MHz, to hundreds of MHz, to GHz, the performance of computer software has taken a back seat to other concerns. Today, it is not at all uncommon for software engineers to extend this maxim to "you should never optimize your code!" Funny, you don't hear too many computer application users making such statements. It is unfortunate that Hoare's comments have been twisted to imply that optimization is unnecessary. The bloat and unresponsiveness found in many modern applications compels software engineers to reconsider how they apply Hoare's comments to their projects.

As an interesting side-node, using the Passmark scores, a top-desktop processor in 2017 is 12 times faster than a similar desktop processor of the same family from 2009! Based on this methodology, our technique of allowing the C compiler’s optimization to do the heavy lifting is not a bad one.

The following is “kstdlib.h”, and contains the prototypes for all of the standard library functions that were used in the rest of this project. If you are not sure about the functionality required by each of these functions you can consult the standard C library documentation on the internet. Implement them in simple C language - none of them are very complex.

|  |
| --- |
| #ifndef \_KSTDLIB\_H  #define \_KSTDLIB\_H  #include <stdint.h>  #ifndef NULL  #define NULL (0)  #endif  typedef uint32\_t size\_t;  void kmemset(void \*ptr, int c, size\_t);  void \*kmemcpy(void \*dest, const void \*src, size\_t n);  void kstrncpy(char \*dest, const char \*src, size\_t n);  int kstrcmp(const char \*a, const char \*b);  int kstrncmp(const char \*a, const char \*b, int n);  int isdigit(char ch);  int islower(char ch);  int isupper(char ch);  int toupper(char ch);  int tolower(char ch);  long int kstrtol(const char \*nptr, char \*\*endptr, int base);  int katoi(const char \*nptr);  // for now, simply move your existing putc and printf here  int kprintf(const char \*format, ...);  void kputc(char ch);  #endif |

### Heap, Malloc, and Free

One of the most complex standard functions provided by the C compiler is its heap management function. In a user application, the heap can grow and shrink dynamically - when the application needs more member it can request that from the OS. More advanced operating systems can also use dynamically allocated and virtual memory to perform the same trick.

A more simple implementation is to use a fixed-size heap that can neither grow nor shrink. As memory is requested through “kmalloc” it will be allocated by the heap from this reserved section of memory. If the section is used up then the allocation will fail.

The simplest way to manage this is to get a pointer to a large, contiguous area of memory. The heap will need to keep track of what has been allocated and freed. The simplest method is to add a prefix to each allocated area. The prefix keeps track of the allocation history. Because this prefix is of fixed size it is easy to determine the prefix’s location given a heap address or a heap address given a prefix’s location.

To protect the heap’s structure, pointers are allocated a “opaque” pointers - their internal fields are not visible outside the heap.

|  |
| --- |
| // opaque pointers  struct knode;  typedef struct knode {  uint32\_t size;  struct knode \*prev, \*next;  } knode\_t; |

This prefix effective tracks each allocation on a doubly-linked list. Because we track both allocations and free regions of memory, there are actually two lists to track: a free regions and an allocated regions.

To fulfill its request, malloc must find a region on the free-list large enough to hold the size allocated and the prefix. This involves traversing the free-regions list and finding a node that can hold the request. The node must be *split****:***

1. If the node is consumed entirely by the request, it is simple deleted from the free list and inserted into the allocated list
2. Else the node is split - a new node pointer is created at the end of the allocated region. The first part of the node will be allocated, so that node is deleted from the list, and the new residual node is inserted into the free list. The allocate node is inserted into the allocated list.

Initial (1MB heap) starting at memory address 10,000. The sizeof(node) is 12 bytes (3 integers)

Free Regions: Allocated Regions: Null

Node 0 @ mem[10000]

|  |  |  |
| --- | --- | --- |
| size:1048576 | Prev: null | Next: null |

**To malloc(1024):**

The first free node can hold the space, so it is split, into a node to hold 1024 + 12 bytes = 1036, and a node that holds the remaining 1,047,540 bytes:  
  
Split:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Node 0: @ mem[10000]   |  |  |  | | --- | --- | --- | | Size: 1036 | Prev: null | Next: null | |  | Node 1: @ mem[11036]   |  |  |  | | --- | --- | --- | | Size: 1047540 | Prev: null | Next: null | |

Lists:

Allocated - points to node 0: Free now points to node 1:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Node 0: @ mem[10000]   |  |  |  | | --- | --- | --- | | Size: 1036 | Prev: null | Next: null | |  | Node 1: @ mem[11036]   |  |  |  | | --- | --- | --- | | Size: 1047540 | Prev: null | Next: null | |

**To malloc(512):**

Start by scanning the free list. The first free node can hold the space, so it is split, into a node to hold 512 + 12 bytes = 524, and a node that holds the remaining 1,047,016 bytes:  
  
Split:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Node 1: @ mem[11036]   |  |  |  | | --- | --- | --- | | Size: 524 | Prev: null | Next: null |   Insert into allocated list (at end) |  | Node 2: @ mem[11560]   |  |  |  | | --- | --- | --- | | Size: 1047016 | Prev: null | Next: null |   Replace previous node |

Lists:

Allocated - points to node 0: Free now points to node 1:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Node 0: @ mem[10000]   |  |  |  | | --- | --- | --- | | Size: 1036 | Prev: null | Next: 11036 |   Node 0: @ mem[11036]   |  |  |  | | --- | --- | --- | | Size: 524 | Prev: 10000 | Next: null | |  | Node 1: @ mem[11560]   |  |  |  | | --- | --- | --- | | Size: 1047016 | Prev: null | Next: null | |

#### Merging Nodes

One complication will be when freeing a node, the nodes address range must be placed back into the free list. The free list must *merge* fully adjacent regions together. As long as adjacent regions are always merged, then we only need to look to the previous and next nodes to merge. This avoid having to roll-up a whole string of merge compatible nodes at once.

Suppose we have the following free list

Node 0: @ mem[10000]

|  |  |  |
| --- | --- | --- |
| Size: 1036 | Prev: null | Next: 11500 |

Node 0: @ mem[11500]

|  |  |  |
| --- | --- | --- |
| Size: 524 | Prev: 10000 | Next: null |

This describes two regions of free memory that are discontiguous: [10,000...11,035), and [11,500...12,024). Suppose we free the block that is in between [11,035...11,500). Then we would want to be left with one free region from [10,000...12,024). To do this, we have to *merge* the compatible blocks. A mege can result in:

* No merge - the inserted block was incompatible with the previous and next, just insert the new one into position
* A left-only merge - the inserted block starts where the previous block ends, so simply update the “size” of the free left block, don’t change anything else, don’t insert the new free block.
* A right-only merge - the right block must be deleted from the list, the new block must be inserted into the list, and its size must be the total of the deleted region and the newly freed chunk
* A left-right merge - Increase the size of the left node’s region to the total of the left, right, and newly freed block. Do not insert the newly freed block. Delete the right block, but change the left node’s “next” pointer to point to the right block’s “next” and make the right block’s “next” node’s previous pointer point at the left node.

There are some obvious corner cases regarding the head and tail of the list.

After adding a free node from 11036 to 11499 (as described above), the free list would be merged into one node with all of the free space *coalesced* into one region:

Node 0: @ mem[10000]

|  |  |  |
| --- | --- | --- |
| Size: 2024 | Prev: null | Next: null |

#### Allocating the Heap

To allocate space for the heap we can use the Linker’s memory map. Edit the “linker.ld” file and add the following:

|  |
| --- |
| .heap BLOCK(4K) : ALIGN(4K)  {  \_heap\_start = ABSOLUTE(.) ;  . += 1M;  \_heap\_end = \_heap\_start + 1M;  \_heap\_size = (\_heap\_end - \_heap\_start);  KEEP(\*(.heap))  } |

This will create symbols \_heap\_start, \_heap\_end, and \_heap\_size. The “.” operator refers to the current address that the linker is using to assign sections. \_heap\_start will be at the exact start of this section. Then, we move the “.” operator forward by 1MB, and the \_heap\_end will be where the “.” has been moved to. Finally, we can get the value for the size of the heap by subtracting the two symbols. The heap will be 1MB.

Remember, these symbols aren’t variables but are replaced with the numeric value they represent. If the \_heap\_start were at memory address 10,000 that is what the symbol would resolve to. We need to have the C compiler work with the memory address at 10,000 - so it will be a pointer as follows.

To aid in this part of the process, I’ve provided my “kmalloc.c” but with some sections of code deleted and others remaining. You should implement the missing sections of code. Remember, this is basically just a sorted, doubly-linked list. You can also extract this code into a separate project and run it on a regular PC using the regular C compiler and debugging tools, then move it back to the kernel and fix any issues from the move. That is actually what I did.

|  |
| --- |
| #include “kmalloc.h”  #ifdef DEBUG  #define PRINTF(...) fprintf(stderr,\_\_VA\_ARGS\_\_)  #else  #define PRINTF(...)  #endif  // defined by the linker  extern uint32\_t \*\_heap\_size;  extern uint8\_t \* \_heap\_start;  extern uint8\_t \* \_heap\_end;  // opaque pointers  struct knode;  typedef struct knode {  uint32\_t size;  struct knode \*prev, \*next;  } knode\_t;  // allocated and free list head nodes  static knode\_t \*free\_list;  static knode\_t \*alloc\_list;  // Initialize the heap  void kmalloc\_init( )  // find the first node that has at least size bytes free  static knode\_t \*kmalloc\_find\_bysize(knode\_t \*list, size\_t size)  {  }  // find the node that immediately precedes this node in the list  // by address  static knode\_t \*kmalloc\_find\_byaddr(knode\_t \*list, knode\_t \*node) {  }  // delete the node from the list  static void kmalloc\_delete\_node(knode\_t \*\*list, knode\_t \*node) {  }  // insert the node at the right position in the list  static void kmalloc\_insert(knode\_t \*\*list, knode\_t \*node) {  }  // split-off size bytes from the src node, return the remaining node  static knode\_t \*kmalloc\_split(knode\_t \*src, size\_t size)  { }  #ifdef DEBUG // print a node to the screen using the current printf function  static void kmalloc\_print\_node(knode\_t \*node)  {  if (node == NULL) {  PRINTF("[NULL]\n");  }  else {  uint8\_t \*ptr = ((uint8\_t \*)node) + 24;  uint8\_t \*end = ((uint8\_t \*)node)+node->size - 1;  PRINTF("[%p/%p: len: %u %p/%p %p)\n", node, ptr, node->size, node->prev, node->next, end);  }  }  // show all of the regions on the allocated and free lists  void kmalloc\_debug\_walk( )  {  knode\_t \*curr;    PRINTF("----------- Allocated Nodes -------------\n");  curr = alloc\_list;  while (curr != NULL) {  kmalloc\_print\_node(curr);  curr = curr->next;  }  PRINTF("\n");  PRINTF("----------- Free Nodes -------------\n");  curr = free\_list;  while (curr != NULL) {  kmalloc\_print\_node(curr);  curr = curr->next;  }  PRINTF("------------------------------------\n");  }  #endif  void \*kmalloc(size\_t size)  {  size\_t internal\_size = size + sizeof(knode\_t);    knode\_t \*free\_node = kmalloc\_find\_bysize(free\_list, internal\_size);  if (free\_node == NULL) return NULL;  kmalloc\_delete\_node(&free\_list, free\_node);  // the free-node will be split, and it will become  // the used space  knode\_t \*remaining = kmalloc\_split(free\_node, internal\_size);  if (remaining->size > 0)  kmalloc\_insert(&free\_list, remaining);  if (alloc\_list == NULL)  alloc\_list = free\_node;  else  kmalloc\_insert(&alloc\_list, free\_node);  return (void \*)(((uint8\_t \*) free\_node)+sizeof(knode\_t));  }  // merge the node ‘region’ into the list. It **should** go into  //the list betwen *prev* and *next*.  static void kmalloc\_merge\_into(knode\_t \*region, knode\_t \*prev, knode\_t \*next)  {  }  // free the block  void kfree(void \*block)  {  knode\_t \*alloc\_node = (knode\_t \*) (((uint8\_t \*) block) - sizeof(knode\_t));  #ifdef DEBUG  kmalloc\_print\_node(alloc\_node);  #endif    // delete the allocated node from the alloc list  kmalloc\_delete\_node(&alloc\_list, alloc\_node);  alloc\_node->prev = alloc\_node->next = NULL; // clear pointers  // put the allocated space back on the free list  // prepare to insert the node into the free list  if (alloc\_node < free\_list){  kmalloc\_merge\_into(alloc\_node, NULL, free\_list);  free\_list = alloc\_node;  }  else {  knode\_t \*prev = kmalloc\_find\_byaddr(free\_list, alloc\_node);  knode\_t \*next = prev->next;  kmalloc\_merge\_into(alloc\_node, prev, next);  } |

### Assembly Intrinsics

The C compiler provides *intrinsic* operators through a library that isn’t available for our bare-metal system. What follows is a header file that implements the most common assembly intrinsics. We’ve actually already used some of these by implementing them ourselves (e.g. in the console). Replace that trash with these standard implementations.

|  |
| --- |
| #ifndef \_386\_H  #define \_386\_H  /\*\*  I386 intrinsic instructions. Much of this is based on code from the  JOS project:  URL: https://github.com/lewischeng-ms/mit-jos/blob/master/inc/x86.h  \*/  // Read a byte from the given input port  static \_\_inline uint8\_t inb(uint16\_t port) {  uint8\_t data;  \_\_asm \_\_volatile("inb %w1, %0" : "=a" (data) : "d" (port));  return data;  }  // Read a 16-bit word from the given input port  static \_\_inline uint16\_t inw(uint16\_t port)  {  uint16\_t data;  \_\_asm \_\_volatile("inw %w1,%0" : "=a" (data) : "d" (port));  return data;  }  // Read a 32-bit long-word from the given input port  static \_\_inline uint32\_t inl(uint16\_t port)  {  uint32\_t data;  \_\_asm \_\_volatile("inl %w1,%0" : "=a" (data) : "d" (port));  return data;  }  // Write a byte to the given output port  static \_\_inline void outb(uint16\_t port, uint8\_t value)  {  \_\_asm \_\_volatile ("outb %0, %w1" : : "a" (value), "d" (port) );  }  // write cnt bytes starting at addr to the port  static \_\_inline void outsb(uint16\_t port, const void \*addr, int cnt)  {  \_\_asm \_\_volatile("cld\n\trepne\n\toutsb" :  "=S" (addr), "=c" (cnt) :  "d" (port), "0" (addr), "1" (cnt) :  "cc");  }  // Invalidate a page identified by address  static \_\_inline void invlpg(void \*addr)  {  \_\_asm \_\_volatile("invlpg (%0)" : : "r" (addr) : "memory");  }  // Load interrupt descriptor table  static \_\_inline void lidt(void \*p)  {  \_\_asm \_\_volatile("lidt (%0)" : : "r" (p));  }  // Load global descriptor table  static \_\_inline void lgdt(void \*p)  {  \_\_asm \_\_volatile("lgdt (%0)" : : "r" (p));  }  // Load local descriptor table  static \_\_inline void lldt(uint16\_t sel)  {  \_\_asm \_\_volatile("lldt %0" : : "r" (sel));  }  // Load task register  static \_\_inline void ltr(uint16\_t sel)  {  \_\_asm \_\_volatile("ltr %0" : : "r" (sel));  }  // Load configuration register 0  static \_\_inline void lcr0(uint32\_t val)  {  \_\_asm \_\_volatile("movl %0,%%cr0" : : "r" (val));  }  // Read configuration register 0  static \_\_inline uint32\_t rcr0(void)  {  uint32\_t val;  \_\_asm \_\_volatile("movl %%cr0,%0" : "=r" (val));  return val;  }  // Read configuration register 2  static \_\_inline uint32\_t rcr2(void)  {  uint32\_t val;  \_\_asm \_\_volatile("movl %%cr2,%0" : "=r" (val));  return val;  }  // Load configuration register 3  static \_\_inline void lcr3(uint32\_t val)  {  \_\_asm \_\_volatile("movl %0,%%cr3" : : "r" (val));  }  // Read configuration register 3  static \_\_inline uint32\_t rcr3(void)  {  uint32\_t val;  \_\_asm \_\_volatile("movl %%cr3,%0" : "=r" (val));  return val;  }  // Load configuration register 4  static \_\_inline void lcr4(uint32\_t val)  {  \_\_asm \_\_volatile("movl %0,%%cr4" : : "r" (val));  }  // Read configuration register 4  static \_\_inline uint32\_t rcr4(void)  {  uint32\_t cr4;  \_\_asm \_\_volatile("movl %%cr4,%0" : "=r" (cr4));  return cr4;  }  // Flush the TLB  static \_\_inline void tlbflush(void)  {  uint32\_t cr3;  \_\_asm \_\_volatile("movl %%cr3,%0" : "=r" (cr3));  \_\_asm \_\_volatile("movl %0,%%cr3" : : "r" (cr3));  }  // Read the EFLAGS register  static \_\_inline uint32\_t read\_eflags(void)  {  uint32\_t eflags;  \_\_asm \_\_volatile("pushfl; popl %0" : "=r" (eflags));  return eflags;  }  // Write to the EFLAGS register  static \_\_inline void write\_eflags(uint32\_t eflags)  {  \_\_asm \_\_volatile("pushl %0; popfl" : : "r" (eflags));  }  // Read EPB register  static \_\_inline uint32\_t read\_ebp(void)  {  uint32\_t ebp;  \_\_asm \_\_volatile("movl %%ebp,%0" : "=r" (ebp));  return ebp;  }  // Read ESP register  static \_\_inline uint32\_t read\_esp(void)  {  uint32\_t esp;  \_\_asm \_\_volatile("movl %%esp,%0" : "=r" (esp));  return esp;  }  // Load CPUID instruction data into given values  static \_\_inline void cpuid(uint32\_t info, uint32\_t \*eaxp,  uint32\_t \*ebxp, uint32\_t \*ecxp,  uint32\_t \*edxp)  {  uint32\_t eax, ebx, ecx, edx;  asm volatile("cpuid"  : "=a" (eax), "=b" (ebx), "=c" (ecx), "=d" (edx)  : "a" (info));  if (eaxp)  \*eaxp = eax;  if (ebxp)  \*ebxp = ebx;  if (ecxp)  \*ecxp = ecx;  if (edxp)  \*edxp = edx;  }  // Read the system's Time Stamp Counter  static \_\_inline uint64\_t read\_tsc(void)  {  uint64\_t tsc;  \_\_asm \_\_volatile("rdtsc" : "=A" (tsc));  return tsc;  }  // Atomically exchange values in memory  static inline uint32\_t xchg(volatile uint32\_t \*addr, uint32\_t newval)  {  uint32\_t result;    // The + in "+m" denotes a read-modify-write operand.  asm volatile("lock; xchgl %0, %1" :  "+m" (\*addr), "=a" (result) :  "1" (newval) :  "cc");  return result;  }  // Disable interrupts  static inline void cli( )  {  asm volatile ("cli;\n");  }  static inline void sti( )  {  asm volatile ("sti;\n");  }  #endif |

### Standard Linked List

It is handy to have a singly linked list for storing data. Now that we have a heap it is easy to implement. Here is my header file, you should implement the C code to match:

|  |
| --- |
| #ifndef \_LIST\_H  #define \_LIST\_H  // should be opaque, didn’t do it  typedef struct list\_node {  void \*data;  struct list\_node \*next;  } list\_node\_t;  typedef struct list {  list\_node\_t \*head;  list\_node\_t \*tail;  int count;  } list\_t;  // initialize the list\_t structure  void list\_init(list\_t \*list);  // kmalloc a node, wrap the data pointer with  // the list node, append on tail. Do not copy  // the data pointer  void list\_append(list\_t \*list, void \*data);  // find the node that contains the data pointer  // remove the node from the chain, kfree the  // node, do not kfree the data pointer  void list\_remove(list\_t \*list, void \*data);  // kmalloc a node, wrap the data pointer with the  // node, insert at the front of the list. Do not  // copy the data pointer  void list\_insert(list\_t \*list, void \*data);  // Iterate over the node, call the handler function on  // each data node in the list - pass the data not the node itself  void list\_iterate(list\_t \*list, int(\*handler)(const void \*));  // Search the list - iterate over each node in the list and  // call the matcher function giving the item being sought after  // and the current list node’s data pointer. Stop the search  // when the matcher returns true and return the current node’s  // data pointer.  void \*list\_search(list\_t \*list, const void \*lookfor, int(\*matcher)(const void \*lookfor, const void \*data));  #endif |

### Bounded FIFO

Another important data structure is the bounded FIFO. We’ve seen this in other places. Here is the header file for my FIFO:

|  |
| --- |
| #ifndef \_FIFO\_H  #define \_FIFO\_H  #include "kstdlib.h"  // create an opaque pointer to the fifo structure  struct fifo;  typedef struct fifo fifo\_t;  // return the size of a fifo node (since its opaque  // we cannot use sizeof  int fifo\_sizeof( );  // initialize the fifo to hold max\_entries, each entry of  // entry size. To construct a fifo with 100 byte entries we would  // do: fifo\_init(&my\_fifo, 100, 1);  // the space to hold the entries is *kmalloc*’d by init  void fifo\_init(fifo\_t \*fifo, size\_t max\_entries, size\_t entry\_size);  // Add an item to the fifo (at the end0  void fifo\_add(fifo\_t \*fifo, void \*entry);  // Get, remove, and return the item at the front  // uses memcpy  void fifo\_get(fifo\_t \*fifo, void \*entry);  // Return true if there is data to be read  int fifo\_avail(fifo\_t \*fifo);  // Return the numeric size of the fifo  int fifo\_size(fifo\_t \*fifo);  // return true if the fifo is full (cannot handle more writes)  int fifo\_full(fifo\_t \*fifo);  // return true if the fifo is empty (cannot handle more reads)  int fifo\_empty(fifo\_t \*fifo);  #endif |

# Deliverables and Demos

Arrange a time for us to meet, and show be prepared to show me the following:

1. While I want to see your code, what I really want is for you to convince me that your code *works*, and the best way for that is to implement the “abstract data types” outside of the kernel and verify that they work with Valgrind, gdb, etc. Write programs that stress-test your code.
2. Then I want to see how you’ve integrated these functions into the project – again, I want to see really good organization and technique.

Points: \_\_\_\_\_\_\_\_\_\_\_ / 50

1. https://ubiquity.acm.org/article.cfm?id=1513451 [↑](#footnote-ref-1)