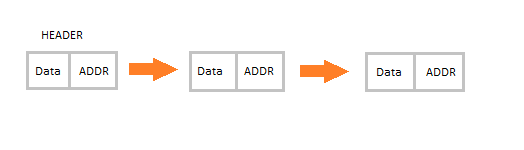
# Part 17 – Linked List in Linux Kernel Part 1

[ <https://embetronicx.com/tutorials/linux/device-drivers/linux-device-driver-tutorial-part-17-linked-list-in-linux-kernel/> ]

This is the [Series on Linux Device Driver](https://www.embetronicx.com/tag/device-driver/). The aim of this series is to provide the easy and practical examples that anyone can understand. In our previous [tutorials](https://www.embetronicx.com/tag/workqueue/) we have seen  work queue. So this is the Linux Device Driver Tutorial Part 17 – Linked List in Linux Kernel Part 1.

# Introduction about Linked List

A linked list is a data structure that consists of sequence of nodes. Each node is composed of two fields: **data field** and **reference field** which is a [pointer](https://en.wikipedia.org/wiki/Pointer_(computer_programming)) that points to the next node in the sequence.



Each node in the list is also called an element. A **head** pointer is used to track the first element in the linked list, therefore, it always points to the first element.

The elements do not necessarily occupy contiguous regions in memory and thus need to be linked together (each element in the list contains a pointer to the *next* element).

Each node in the list is also called an element. A **head** pointer is used to track the first element in the linked list, therefore, it always points to the first element.

The elements do not necessarily occupy contiguous regions in memory and thus need to be linked together (each element in the list contains a pointer to the next element).

## Advantages of Linked Lists

* They are a dynamic in nature which allocates the memory when required.
* Insertion and deletion operations can be easily implemented.
* Stacks and queues can be easily executed.
* Linked List reduces the access time.

## Disadvantages of Linked Lists

* The memory is wasted as pointers require extra memory for storage.
* No element can be accessed randomly; it has to access each node sequentially.
* Reverse Traversing is difficult in linked list.

## Applications of Linked Lists

* Linked lists are used to implement stacks, queues, graphs, etc.
* Unlike array, In Linked Lists we don’t need to know the size in advance.

# Types of Linked Lists

There are three types of linked lists.

* Singly Linked List
* Doubly Linked List
* Circular Linked List

Let’s get into the Linked List in Linux kernel.

# Linked List in Linux Kernel

Linked list is a very important data structure which allows large number of storage with efficient manipulation on data. Most of the kernel code has been written with help of this data structure. So in Linux kernel no need to implement our own Linked List or no need to use 3rd party library.

It has built in Linked List which is Doubly Linked List. It is defined in defined in /lib/modules/$(uname -r)/build/include/linux/list.h.

Normally user-defined linked list is defined as below snippet.

// User Defined Implementation

struct my\_list{

int data,

struct my\_list \*prev;

struct my\_list \*next;

};

But for implementing LinkedList in Linux, then you could write like below snippet.

// Linux Implementation of Linked List

struct my\_list{

struct list\_head list;

int data,

};

Where struct list\_head is declared in list.h.

struct list\_head {

     struct list\_head \*next;

     struct list\_head \*prev;

};

# Initialize Linked List Head

Before creating any node in linked list, we should create linked list’s head node first. So below macro is used to create a head node.

LIST\_HEAD(linked\_list);

This macro will create the head node structure in the name of “linked\_list” and it will initialize that to its own address.

For example,

I’m going to create head node in the name of “etx\_linked\_list”.

LIST\_HEAD(etx\_linked\_list);

Let’s see how internally it handles this. The macro is defined like below in list.h.

#define LIST\_HEAD\_INIT(name) { &(name), &(name) }

#define LIST\_HEAD(name) \

struct list\_head name = LIST\_HEAD\_INIT(name)

struct list\_head {

struct list\_head \*next;

struct list\_head \*prev;

};

So it will create like below.

struct list\_head etx\_linked\_list = {

&etx\_linked\_list ,

&etx\_linked\_list

};

While creating head node, it initializes the prev and next pointer to its own address. Which means that prev and next pointer points to itself. The list is empty If the node’s prev and next pointer points to itself.

# Create Node in Linked List

You have to create your linked list node dynamically or statically. Your linked list node should have member defined in struct list\_head. Using below inline function, we can initialize that struct list\_head.

INIT\_LIST\_HEAD(struct list\_head \*list);

For Example, My node is like this.

struct my\_list{

     struct list\_head list;     //linux kernel list implementation

     int data;

};

struct my\_list new\_node;

So we have to initialize the list\_head variable using INIT\_LIST\_HEAD inline function.

INIT\_LIST\_HEAD(&new\_node.list);

new\_node.data = 10;

# Add Node to Linked List

## Add after Head Node

After created that node, we need to add that node to the linked list. So we can use this inline function to do that.

inline void list\_add(struct list\_head \*new, struct list\_head \*head);

Insert a new entry **after the specified head**. This is good for implementing stacks.

Where,

struct list\_head \* new – new entry to be added

struct list\_head \* head – list head to add it after

For Example,

**list\_add**(&new\_node.list, &etx\_linked\_list);

## Add before Head Node

|  |
| --- |
| Insert a new entry before the specified head. This is useful for implementing queues.  inline void list\_add\_tail(struct list\_head \*new, struct list\_head \*head);  Where,  struct list\_head \* new – new entry to be added  struct list\_head \* head – list head to add before the head  For Example,  **list\_add\_tail**(&new\_node.list, &etx\_linked\_list); |

# Delete Node from Linked List

## list\_del

|  |
| --- |
| It will delete the entry node from the list. This function removes the entry node from the linked list by disconnect prev and next pointers from the list, but it doesn’t free any memory space allocated for entry node.  inline void list\_del(struct list\_head \*entry);  Where,  struct list\_head \* entry– the element to delete from the list. |

## list\_del\_init

|  |
| --- |
| It will delete the entry node from the list and reinitialize it. This function removes the entry node from the linked list by disconnect prev and next pointers from the list, but it doesn’t free any memory space allocated for entry node.  inline void list\_del\_init(struct list\_head \*entry);  Where,  struct list\_head \* entry– the element to delete from the list. |

# Replace Node in Linked List

## list\_replace

|  |
| --- |
| This function is used to replace the old node with new node.  inline void list\_replace(struct list\_head \*old, struct list\_head \*new);  Where,  struct list\_head \* old– the element to be replaced  struct list\_head \* new– the new element to insert  If *old* was empty, it will be overwritten. |

## list\_replace\_init

|  |
| --- |
| This function is used to replace the old node with new node and reinitialize the old entry.  inline void list\_replace\_init(struct list\_head \*old, struct list\_head \*new);  Where,  struct list\_head \* old– the element to be replaced  struct list\_head \* new– the new element to insert  If *old* was empty, it will be overwritten. |

# Moving Node in Linked List

## list\_move

|  |
| --- |
| This will delete one list node from the linked list and again adds to after the head node.  inline void list\_move(struct list\_head \*list, struct list\_head \*head);  Where,  struct list\_head \* list – the entry to move  struct list\_head \* head– the head that will precede our entry |

## list\_move\_tail

|  |
| --- |
| This will delete one list from the linked list and again adds to before the head node.  inline void list\_move\_tail(struct list\_head \*list, struct list\_head \*head);  Where,  struct list\_head \* list – the entry to move  struct list\_head \* head– the head that will precede our entry |

# Rotate Node in Linked List

|  |
| --- |
| This will rotate the list to the left.  inline void list\_rotate\_left(struct list\_head \*head);  Where,  head – the head of the list |

# Test the Linked List Entry

## list\_is\_last

|  |
| --- |
| This tests whether *list* is the last entry in list *head*.  inline int list\_is\_last(const struct list\_head \*list,  const struct list\_head \*head);  Where,  const struct list\_head \* list – the entry to test  const struct list\_head \* head – the head of the list  It returns**1** if it is last entry otherwise **0**. |

## list\_empty

|  |
| --- |
| It tests whether a list is empty or not.  inline int list\_empty(const struct list\_head \*head);  Where,  const struct list\_head \* head – the head of the list  It returns**1** if it is empty otherwise **0**. |

## list\_is\_singular

|  |
| --- |
| This will tests whether a list has just one entry.  inline int list\_is\_singular(const struct list\_head \*head);  Where,  const struct list\_head \* head – the head of the list  It returns**1** if it has only one entry otherwise **0**. |

# Split Linked List into two part

|  |
| --- |
| This cut a list into two.  This helper moves the initial part of *head*, up to and including *entry*, from *head* to *list*. You should pass on *entry* an element you know is on *head*. *list* should be an empty list or a list you do not care about losing its data.  inline void list\_cut\_position(struct list\_head \*list,  struct list\_head \*head,  struct list\_head \*entry);  Where,  struct list\_head \* list – a new list to add all removed entries  struct list\_head \* head– a list with entries  struct list\_head \* entry– an entry within head, could be the head itself and if so we won’t cut the list |

# Join Two Linked Lists

|  |
| --- |
| This will join two lists, this is designed for stacks.  inline void list\_splice(const struct list\_head \*list, struct list\_head \*head);  Where,  const struct list\_head \* list – the new list to add.  struct list\_head \* head – the place to add it in the first list. |

# Traverse Linked List

## list\_entry

|  |
| --- |
| This macro is used to get the struct for this entry.  list\_entry(ptr, type, member);  ptr – the struct list\_head pointer.  type – the type of the struct this is embedded in.  member – the name of the list\_head within the struct. |

## list\_for\_each

|  |
| --- |
| This macro used to iterate over a list.  list\_for\_each(pos, head);  pos –  the &struct list\_head to use as a loop cursor.  head –  the head for your list. |

So using those above two macros, we can traverse the linked list.

## list\_for\_each\_entry

|  |
| --- |
| This is used to iterate over list of given type.  list\_for\_each\_entry(pos, head, member);  pos – the type \* to use as a loop cursor.  head – the head for your list.  member – the name of the list\_head within the struct. |

## list\_for\_each\_entry\_safe

|  |
| --- |
| This will iterate over list of given type safe against removal of list entry.  list\_for\_each\_entry\_safe ( pos, n, head, member);  Where,  pos – the type \* to use as a loop cursor.  n – another type \* to use as temporary storage  head – the head for your list.  member – the name of the list\_head within the struct. |

We can also **traverse the linked list in reverse side** also using below macros.

## list\_for\_each\_prev

|  |
| --- |
| This will used to iterate over a list backwards.  list\_for\_each\_prev(pos, head);  pos – the &struct list\_head to use as a loop cursor.  head – the head for your list. |

## list\_for\_each\_entry\_reverse

|  |
| --- |
| This macro used to iterate backwards over list of given type.  list\_for\_each\_entry\_reverse(pos, head, member);  pos – the type \* to use as a loop cursor.  head the head for your list.  member – the name of the list\_head within the struct. |

So, We have gone through all the functions which is useful for Kernel Linked List. Please go through the next tutorial ([Part 2](https://www.embetronicx.com/tutorials/linux/device-drivers/example-linked-list-in-linux-kernel/)) for Linked List sample program .