

Libraries

COLLABORATORS

	TITLE : Libraries		
ACTION	NAME	DATE	SIGNATURE
WRITTEN BY		July 23, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Libraries	1
1.1	Amiga® RKM Libraries: 23 Exec Lists and Queues	1
1.2	23 Exec Lists and Queues / List Structure	1
1.3	23 / List Structure / Node Structure Definition	2
1.4	23 / List Structure / Node Initialization	3
1.5	23 / List Structure / List Header Structure Definition	4
1.6	23 / List Structure / Header Initialization	5
1.7	23 Exec Lists and Queues / List Functions	6
1.8	23 / List Functions / Insertion and Removal	7
1.9	23 / List Functions / Special Case Insertion	7
1.10	23 / List Functions / Special Case Removal	7
1.11	23 / List Functions / MinList/MinNode Operations	8
1.12	23 / List Functions / Prioritized Insertion	8
1.13	23 / List Functions / Searching by Name	8
1.14	23 / List Functions / More on the Use of Named Lists	9
1.15	23 / List Functions / List Macros for Assembly Language Programmers	9
1.16	23 / List Functions / Empty Lists	10
1.17	23 / List Functions / Scanning a List	10
1.18	23 / List Functions / Important Note About Shared Lists	11
1.19	23 Exec Lists and Queues / Function Reference	12

Chapter 1

Libraries

1.1 Amiga® RKM Libraries: 23 Exec Lists and Queues

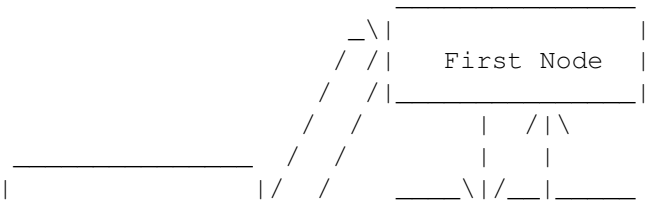
The Amiga system software operates in a dynamic environment of data structures. An early design goal of Exec was to keep the system flexible and open-ended by eliminating artificial boundaries on the number of system structures used. Rather than using static system tables, Exec uses dynamically created structures that are added and removed as needed. These structures can be put in an unordered list, or in an ordered list known as a queue. A list can be empty, but never full. This concept is central to the design of Exec. Understanding lists and queues is important to understanding not only Exec itself, but also the mechanism behind the Amiga's message and port based interprocess communication.

Exec uses lists to maintain its internal database of system structures. Tasks, interrupts, libraries, devices, messages, I/O requests, and all other Exec data structures are supported and serviced through the consistent application of Exec's list mechanism. Lists have a common data structure, and a common set of functions is used for manipulating them. Because all of these structures are treated in a similar manner, only a small number of list handling functions need be supported by Exec.

List Structure List Functions Function Reference

1.2 23 Exec Lists and Queues / List Structure

A list is composed of a header and a doubly-linked chain of elements called nodes. The header contains memory pointers to the first and last nodes of the linked chain. The address of the header is used as the handle to the entire list. To manipulate a list, you must provide the address of its header.



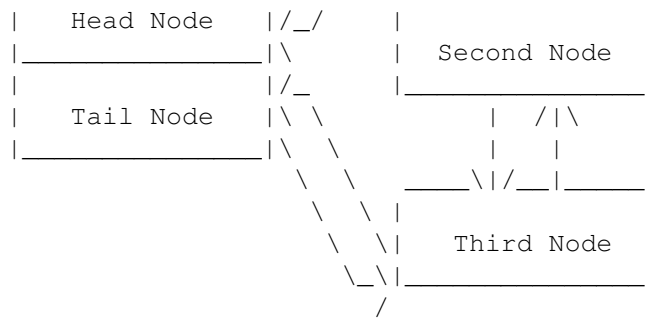


Figure 23-1: Simplified Overview of an Exec List

Nodes may be scattered anywhere in memory. Each node contains two pointers; a successor and a predecessor. As illustrated above, a list header contains two placeholder nodes that contain no data. In an empty list, the head and tail nodes point to each other.

Node Structure Definition
Node Initialization

List Header Structure Definition
Header Initialization

1.3 23 / List Structure / Node Structure Definition

A Node structure is divided into three parts: linkage, information, and content. The linkage part contains memory pointers to the node's successor and predecessor nodes. The information part contains the node type, the priority, and a name pointer. The content part stores the actual data structure of interest. For nodes that require linkage only, a small MinNode structure is used.

```
struct MinNode
{
    struct MinNode *mln_Succ;
    struct MinNode *mln_Pred;
};
```

ln_Succ
points to the next node in the list (successor).

ln_Pred
points to the previous node in the list (predecessor).

When a type, priority, or name is required, a full-featured Node structure is used.

```
struct Node
{
    struct Node *ln_Succ;
    struct Node *ln_Pred;
    UBYTE      ln_Type;
    BYTE       ln_Pri;
    char       *ln_Name;
};
```

`ln_Type`
 defines the type of the node (see `<exec/nodes.h>` for a list).

`ln_Pri`
 specifies the priority of the node (+127 (highest) to -128 (lowest)).

`ln_Name`
 points to a printable name for the node (a NULL-terminated string).

The `Node` and `MinNode` structures are often incorporated into larger structures, so groups of the larger structures can easily be linked together. For example, the `Exec Interrupt` structure is defined as follows:

```
struct Interrupt
{
    struct Node is_Node;
    APTR        is_Data;
    VOID        (*is_Code) ();
};
```

Here the `is_Data` and `is_Code` fields represent the useful content of the node. Because the `Interrupt` structure begins with a `Node` structure, it may be passed to any of the `Exec List` manipulation functions.

1.4 23 / List Structure / Node Initialization

Before linking a node into a list, certain fields may need initialization. Initialization consists of setting the `ln_Type`, `ln_Pri`, and `ln_Name` fields to their appropriate values (A `MinNode` structure does not have these fields). The successor and predecessor fields do not require initialization.

The `ln_Type` field contains the data type of the node. This indicates to `Exec` (and other subsystems) the type, and hence the structure, of the content portion of the node (the extra data after the `Node` structure). The standard system types are defined in the `<exec/nodes.h>` include file. Some examples of standard system types are `NT_TASK`, `NT_INTERRUPT`, `NT_DEVICE`, and `NT_MSGPORT`.

The `ln_Pri` field uses a signed numerical value ranging from +127 to -128 to indicate the priority of the node. Higher-priority nodes have greater values; for example, 127 is the highest priority, zero is nominal priority, and -128 is the lowest priority. Some `Exec` lists are kept sorted by priority order. In such lists, the highest-priority node is at the head of the list, and the lowest-priority node is at the tail of the list. Most `Exec` node types do not use a priority. In such cases, initialize the priority field to zero.

The `ln_Name` field is a pointer to a NULL-terminated string of characters. Node names are used to find and identify list-bound objects (like public message ports and libraries), and to bind symbolic names to actual nodes. Names are also useful for debugging purposes, so it is a good idea to

provide every node with a name. Take care to provide a valid name pointer; Exec does not copy name strings.

This fragment initializes a Node called myInt, an instance of the Interrupt data structure introduced above.

```
struct Interrupt interrupt;

interrupt.is_Node.ln_Type = NT_INTERRUPT;
interrupt.is_Node.ln_Pri  = -10;
interrupt.is_Node.ln_Name = "sample.interrupt";
```

1.5 23 / List Structure / List Header Structure Definition

As mentioned earlier, a list header maintains memory pointers to the first and last nodes of the linked chain of nodes. It also serves as a handle for referencing the entire list. The minimum list header ("mlh_") and the full-featured list header ("lh_") are generally interchangeable.

The structure MinList defines a minimum list header.

```
struct MinList
{
    struct MinNode *mlh_Head;
    struct MinNode *mlh_Tail;
    struct MinNode *mlh_TailPred;
};
```

mlh_Head
points to the first node in the list.

mlh_Tail
is always NULL.

mlh_TailPred
points to the last node in the list.

In a few limited cases a full-featured List structure will be required:

```
struct List
{
    struct Node *lh_Head;
    struct Node *lh_Tail;
    struct Node *lh_TailPred;
    UBYTE      lh_Type;
    UBYTE      lh_Pad;
};
```

lh_Type
defines the type of nodes within the list (see <exec/nodes.h>).

lh_pad
is a structure alignment byte.

One subtlety here must be explained further. The list header is

constructed in an efficient, but confusing manner. Think of the header as a structure containing the head and tail nodes for the list. The head and tail nodes are placeholders, and never carry data. The head and tail portions of the header actually overlap in memory. `lh_Head` and `lh_Tail` form the head node; `lh_Tail` and `lh_TailPred` form the tail node. This makes it easy to find the start or end of the list, and eliminates any special cases for insertion or removal.

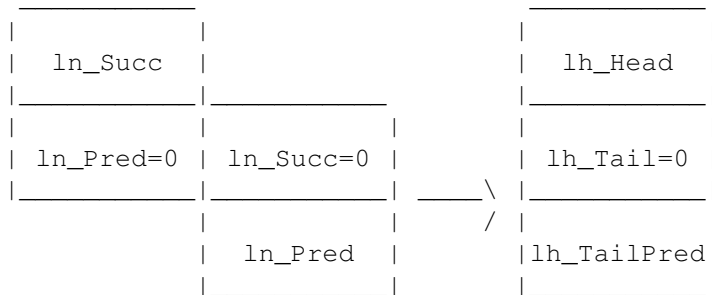


Figure 23-2: List Header Overlap

The `lh_Head` and `lh_Tail` fields of the list header act like the `ln_Succ` and `lh_Pred` fields of a node. The `lh_Tail` field is set permanently to `NULL`, indicating that the head node is indeed the first on the list -- that is, it has no predecessors. See the figure above.

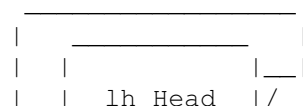
Likewise, the `lh_Tail` and `lh_TailPred` fields of the list header act like the `ln_Succ` and `lh_Pred` fields of a node. Here the `NULL` `lh_Tail` indicates that the tail node is indeed the last on the list -- that is, it has no successors. See the figure above.

1.6 23 / List Structure / Header Initialization

List headers must be properly initialized before use. It is not adequate to initialize the entire header to zero. The head and tail entries must have specific values. The header must be initialized as follows:

1. Set the `lh_Head` field to the address of `lh_Tail`.
2. Clear the `lh_Tail` field.
3. Set the `lh_TailPred` field to the address of `lh_Head`.
4. Set `lh_Type` to the same data type as the nodes to be kept the list. (Unless you are using a `MinList`).

Figure 23-3: Initializing a List Header Structure




```

| | _____ | \ | |
|_ \ | | | |
|/ | lh_Tail=0 | |
| _____ | |
| | | | |
|lh_TailPred|_ _|
|_ _ _ _ _|
| | | | |
/* C example - equivalent to NewList() */
struct List list;
|_ _ _|_ _ _|

list.lh_Head      = (struct Node *) &list.lh_Tail;
list.lh_Tail      = 0;
list.lh_TailPred  = (struct Node*) &list.lh_Head;
/* Now set lh_Type, if needed */

;Assembly example - equivalent to NEWLIST
MOVE.L  A0,LH_HEAD(A0) ;A0 points to the list header
ADDQ.L  #4,LH_HEAD(A0) ;Bump LH_HEAD(A0) to address of LH_TAIL
CLR.L   LH_TAIL(A0)
MOVE.L  A0,LH_TAILPRED(A0)
;Now set LH_TYPE, if needed.

```

The sequence of assembly instructions in the figure above is equivalent to the macro NEWLIST, contained in the include file <exec/lists.i>. Since the MinList structure is the same as the List structure except for the type and pad fields, this sequence of assembly language code will work for both structures. The sequence performs its function without destroying the pointer to the list header in A0 (which is why ADDQ.L is used). This function may also be accessed from C as a call to NewList(header), where header is the address of a list header.

1.7 23 Exec Lists and Queues / List Functions

Exec provides a number of symmetric functions for handling lists. There are functions for inserting and removing nodes, for adding and removing head and tail nodes, for inserting nodes in a priority order, and for searching for nodes by name. The prototypes for Exec list handling functions are as follows.

Exec Functions

```

VOID AddHead( struct List *list, struct Node *node );
VOID AddTail( struct List *list, struct Node *node );
VOID Enqueue( struct List *list, struct Node *node );
struct Node *FindName( struct List *list, UBYTE *name );
VOID Insert( struct List *list, struct Node *node, struct Node *pred );
VOID Remove( struct Node *node );
struct Node *RemHead( struct List *list );
struct Node *RemTail( struct List *list );

```

Exec Support Functions in amiga.lib

```

VOID NewList( struct List *list );

```

In this discussion of the Exec list handling functions, header represents a pointer to List header, and node represents pointer to a Node.

- Insertion and Removal
- Special Case Insertion
- Special Case Removal
- MinList/MinNode Operations
- Prioritized Insertion
- Searching by Name
- More on the Use of Named Lists
- List Macros for Assembly Language Programmers
- Empty Lists
- Scanning a List
- Important Note About Shared Lists

1.8 23 / List Functions / Insertion and Removal

The `Insert()` function is used for inserting a new node into any position in a list. It always inserts the node following a specified node that is already part of the list. For example, `Insert(header,node,pred)` inserts the node `node` after the node `pred` in the specified list. If the `pred` node points to the list header or is `NULL`, the new node will be inserted at the head of the list. Similarly, if the `pred` node points to the `lh_Tail` of the list, the new node will be inserted at the tail of the list. However, both of these actions can be better accomplished with the functions mentioned in the "Special Case Insertion" section below.

The `Remove()` function is used to remove a specified node from a list. For example, `Remove(node)` will remove the specified node from whatever list it is in. To be removed, a node must actually be in a list. If you attempt to remove a node that is not in a list, you will cause serious system problems.

1.9 23 / List Functions / Special Case Insertion

Although the `Insert()` function allows new nodes to be inserted at the head and the tail of a list, the `AddHead()` and `AddTail()` functions will do so with higher efficiency. Adding to the head or tail of a list is common practice in first-in-first-out (FIFO) or last-in-first-out (LIFO or stack) operations. For example, `AddHead(header,node)` would insert the node at the head of the specified list.

1.10 23 / List Functions / Special Case Removal

The two functions `RemHead()` and `RemTail()` are used in combination with `AddHead()` and `AddTail()` to create special list ordering. When you combine `AddTail()` and `RemHead()`, you produce a first-in-first-out (FIFO) list. When you combine `AddHead()` and `RemHead()` a last-in-first-out (LIFO or stack) list is produced. `RemTail()` exists for symmetry. Other

combinations of these functions can also be used productively.

Both `RemHead()` and `RemTail()` remove a node from the list, and return a pointer to the removed node. If the list is empty, the function return a NULL result.

1.11 23 / List Functions / MinList/MinNode Operations

All of the above functions and macros will work with long or short format node structures. A `MinNode` structure contains only linkage information. A full `Node` structure contains linkage information, as well as type, priority and name fields. The smaller `MinNode` is used where space and memory alignment issues are important. The larger `Node` is used for queues or lists that require a name tag for each node.

1.12 23 / List Functions / Prioritized Insertion

The list functions discussed so far do not make use of the priority field in a `Node`. The `Enqueue()` function is equivalent to `Insert()`, except it inserts nodes into a list sorting them according to their priority. It keeps the higher-priority nodes towards the head of the list. All nodes passed to this function must have their priority and name assigned prior to the call. `Enqueue(header,mynode)` inserts `mynode` behind the lowest priority node with a priority greater than or equal to `mynode`'s. For `Enqueue()` to work properly, the list must already be sort according to priority. Because the highest priority node is at the head of the list, the `RemHead()` function will remove the highest-priority node. Likewise, `RemTail()` will remove the lowest-priority node.

FIFO Is Used For The Same Priority.

If you add a node that has the same priority as another node in the queue, `Enqueue()` will use FIFO ordering. The new node is inserted following the last node of equal priority.

1.13 23 / List Functions / Searching by Name

Because many lists contain nodes with symbolic names attached (via the `ln_Name` field), it is possible to find a node by its name. This naming technique is used throughout `Exec` for such nodes as tasks, libraries, devices, and resources.

The `FindName()` function searches a list for the first node with a given name. For example, `FindName(header, "Furrbol")` returns a pointer to the first node named "Furrbol." If no such node exists, a NULL is returned. The case of the name characters is significant; "foo" is different from "Foo."

specified list operation into the code, they result in slightly faster operations. The file <exec/lists.i> contains the recommended set of macros. For example, the following instructions implement the REMOVE macro:

```
MOVE.L LN_SUCC(A1),A0      ; get successor
MOVE.L LN_PRED(A1),A1      ; get predecessor
MOVE.L A0,LN_SUCC(A1)      ; fix up predecessor's succ pointer
MOVE.L A1,LN_PRED(A0)      ; fix up successor's pred pointer
```

1.16 23 / List Functions / Empty Lists

It is often important to determine if a list is empty. This can be done in many ways, but only two are worth mentioning. If either the lh_TailPred field is pointing to the list header or the ln_Succ field of the lh_Head is NULL, then the list is empty.

In C, for example, these methods would be written as follows:

```
/* You can use this method... */
if (list->lh_TailPred == (struct Node *)list)
    printf("list is empty\n");

/* Or you can use this method */
if (NULL == list->lh_Head->ln_Succ)
    printf("list is empty\n");
```

In assembly code, if A0 points to the list header, these methods would be written as follows:

```
; Use this method...
CMP.L LH_TAILPRED(A0),A0
BEQ    list_is_empty

; Or use this method
MOVE.L LH_HEAD(A0),A1
TST.L LN_SUCC(A1)
BEQ    list_is_empty
```

Because LH_HEAD and LN_SUCC are both zero offsets, the second case may be simplified or optimized by your assembler.

1.17 23 / List Functions / Scanning a List

Occasionally a program may need to scan a list to locate a particular node, find a node that has a field with a particular value, or just print the list. Because lists are linked in both the forward and backward directions, the list can be scanned from either the head or tail.

Here is a code fragment that uses a for loop to print the names of all nodes in a list:

```
struct List *list;
```

```

struct Node *node;

for (node = list->lh_Head ; node->ln_Succ ; node = node->ln_Succ)
    printf("%lx -> %s\n",node,node->ln_Name);

```

A common mistake is to process the head or tail nodes. Valid data nodes have non-NULL successor and predecessor pointers. The above loop exits when node->ln_Succ is NULL. Another common mistake is to free a node from within a loop, then reference the free memory to obtain the next node pointer. An extra temporary pointer solves this second problem.

In assembly code, it is more efficient to use a look-ahead cache pointer when scanning a list. In this example the list is scanned until the first zero-priority node is reached:

```

                MOVE.L   (A1),D1           ; first node
scan:           MOVE.L   D1,A1
                MOVE.L   (A1),D1           ; lookahead to next
                BEQ.S     not_found         ; end of list...
                TST.B     LN_PRI(A1)
                BNE.S     scan
                ...
                ; found one

not_found:

Exec List Example

```

1.18 23 / List Functions / Important Note About Shared Lists

It is possible to run into contention problems with other tasks when manipulating a list that is shared by more than one task. None of the standard Exec list functions arbitrates for access to the list. For example, if some other task happens to be modifying a list while your task scans it, an inconsistent view of the list may be formed. This can result in a corrupted system.

Generally it is not permissible to read or write a shared list without first locking out access from other tasks. All users of a list must use the same arbitration method. Several arbitration techniques are used on the Amiga. Some lists are protected by a semaphore. The ObtainSemaphore() call grants ownership of the list (see the "Exec Semaphores" chapter for more information). Some lists require special arbitration. For example, you must use the Intuition LockIBase(0) call before accessing any Intuition lists. Other lists may be accessed only during Forbid() or Disable() (see the "Exec Tasks" chapter for more information).

The preferred method for arbitrating use of a shared list is through semaphores because a semaphore only holds off other tasks that are trying to access the shared list. Rather than suspending all multitasking. Failure to lock a shared list before use will result in unreliable operation.

Note that I/O functions including printf() generally call Wait() to wait for I/O completion, and this allows other tasks to run. Therefore, it is not safe to print or Wait() while traversing a list unless the list is

fully controlled by your application, or if the list is otherwise guaranteed not to change during multitasking.

1.19 23 Exec Lists and Queues / Function Reference

The following charts give a brief description of the Exec list and queue functions and assembler macros. See the Amiga ROM Kernel Reference Manual: Includes and Autodocs for details about each call.

Table 23-1: Exec List and Queue Functions

Exec Function	Description
AddHead()	Insert a node at the head of a list.
AddTail()	Append a node to the tail of a list.
Enqueue()	Insert or append a node to a system queue.
FindName()	Find a node with a given name in a system list.
Insert()	Insert a node into a list.
IsListEmpty	Test if list is empty
NewList()	Initialize a list structure for use.
RemHead()	Remove the head node from a list.
Remove()	Remove a node from a list.
RemTail()	Remove the tail node from a list.

Table 23-2: Exec List and Queue Assembler Macros

Exec Function	Description
NEWLIST	Initialize a list header for use.
TSTLIST	Test if list is empty (list address in register). No arbitration needed.
TSTLIST2	Test if list is empty (from effective address of list). Arbitration needed.
SUCC	Get next node in a list.
PRED	Get previous node in a list.
IFEMPTY	Branch if list is empty.
IFNOTEMPTY	Branch if list is not empty.
TSTNODE	Get next node, test if at end of list.
NEXTNODE	Get next node, go to exit label if at end.
ADDHEAD	Add node to head of list.
ADDTAIL	Add node to tail of list.
REMOVE	Remove node from a list.
REMHEAD	Remove node from head of list.
REMHEADQ	Remove node from head of list quickly.
REMTAIL	Remove node from tail of list.