

Libraries_Manual

COLLABORATORS

	<i>TITLE :</i> Libraries_Manual		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		July 18, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Libraries_Manual	1
1.1	Amiga® RKM Libraries: 37 Utility Library	1
1.2	37 Utility Library / Tags	1
1.3	37 / Tags / Tag Functions and Structures	1
1.4	37 / Tags / Simple Tag Usage	2
1.5	37 / Tags / Advanced Tag Usage	4
1.6	37 // Advanced Tag Usage / Creating a New Tag List	4
1.7	37 // Advanced Tag Usage / Copying an Existing Tag List	4
1.8	37 // Advanced Tag Usage / Filtering an Existing Tag List	5
1.9	37 // Advanced Tag Usage / Locating an Attribute	6
1.10	37 // Advanced Tag Usage / Sequential Access of Tag Lists	7
1.11	37 // Advanced Tag Usage / Random Access of Tag Lists	8
1.12	37 // Advanced Tag Usage / Obtaining Boolean Values	8
1.13	37 // Advanced Tag Usage / Mapping Tag Attributes	8
1.14	37 Utility Library / Callback Hooks	9
1.15	37 / Callback Hooks / Callback Hook Structure and Function	9
1.16	37 // Hook Structure And Function / Simple Callback Hook Usage	10
1.17	37 Utility Library / 32-bit Integer Math Functions	12
1.18	37 Utility Library / International String Functions	13
1.19	37 Utility Library / Date Functions	13
1.20	37 Utility Library / Function Reference	14
1.21	37 / Function Reference / Tag Function Reference	14
1.22	37 / Function Reference / Callback Hook Function Reference	15
1.23	37 / Function Reference / 32-Bit Integer Math Function Reference	15
1.24	37 / Function Reference / International String Function Reference	16
1.25	37 / Function Reference / Date Function Reference	16

Chapter 1

Libraries_Manual

1.1 Amiga® RKM Libraries: 37 Utility Library

Utility library is the home for all the OS functions which don't fit in the other libraries. Among the calls in utility library are calls to manage tags and tag lists, callback hooks, and some generic 32-bit math functions, all discussed below.

Tags	International String Functions
Callback Hooks	Date Functions
32-bit Integer Math Functions	Function Reference

1.2 37 Utility Library / Tags

The implementation of tags is one of the many new features of Release 2. Tags make it possible to add new parameters to system functions without interfering with the original parameters. They also make specifying parameter lists much clearer and easier.

Tag Functions and Structures	Simple Tag Usage Example
Simple Tag Usage	Advanced Tag Usage

1.3 37 / Tags / Tag Functions and Structures

A tag is made up of an attribute/value pair as defined below (from <utility/tagitem.h>):

```
struct TagItem
{
    ULONG   ti_Tag;    /* identifies the type of this item */
    ULONG   ti_Data;   /* type-specific data, can be a pointer */
};
```

The ti_Tag field specifies an attribute to set. The possible values of ti_Tag are implementation specific. System tags are defined in the include files. The value the attribute is set to is specified in ti_Data.

An example of the attribute/value pair that will specify a window's name is:

```
ti_Tag = WA_Title;
ti_Data = "My Window's Name";
```

The ti_Data field often contains 32-bit data as well as pointers.

These are brief descriptions of the utility functions you can use to manipulate and access tags. For complete descriptions, see the "Simple Tag Usage" and "Advanced Tag Usage" sections.

The following utility library calls are for supporting tags:

Table 37-1: Utility Library Tag Functions

AllocateTagItems()	Allocate a TagItem array (or chain).
FreeTagItems()	Frees allocated TagItem lists.
CloneTagItems()	Copies a TagItem list.
RefreshTagItemClones()	Rejuvenates a clone from the original.
FindTagItem()	Scans TagItem list for a tag.
GetTagData()	Obtain data corresponding to tag.
NextTagItem()	Iterate TagItem lists.
TagInArray()	Check if a tag value appears in a Tag array.
FilterTagChanges()	Eliminate TagItems which specify no change.
FilterTagItems()	Remove selected items from a TagItem list.
MapTags()	Convert ti_Tag values in a list via map pairing.
PackBoolTags()	Builds a "Flag" word from a TagItem list.

1.4 37 / Tags / Simple Tag Usage

One way tags are passed to system functions is in the form of tag lists. A tag list is an array or chain of arrays of TagItem structures. Within this array, different data items are identified by the value of ti_Tag. Items specific to a subsystem (Intuition, Graphics,...) have a ti_Tag value which has the TAG_USER bit set. Global system tags have a ti_Tag value with TAG_USER bit clear. The global system tags include:

Table 37-2: Global System Tags

Tag Value	Meaning
TAG_IGNORE	A no-op. The data item is ignored.
TAG_MORE	The ti_Data points to another tag list, to support chaining of TagItem arrays.

TAG_DONE	Terminates the TagItem array (or chain).
TAG_SKIP	Ignore the current tag item, and skip the next n array
	elements, where n is kept in ti_Data.

Note that user tags need only be unique within the particular context of their use. For example, the attribute tags defined for `OpenWindow()` have the same numeric value as some tags used by `OpenScreen()`, but the same numeric value has different meaning in the different contexts.

System functions receive TagItems in several ways. One way is illustrated in the Intuition function `OpenWindow()`. This function supports an extended NewWindow structure called `ExtNewWindow`. When the `NW_EXTENDED` flag is set in the `ExtNewWindow.Flags` field, `OpenWindow()` assumes that the `ExtNewWindow.Extension` field contains a pointer to a tag list.

Another method of passing a tag list is to directly pass a pointer to a tag list, as `OpenWindowTagList()` does in the following code fragment.

```
struct TagItem tagitem[3];
struct Screen *screen;
struct Window *window;

tagitem[0].ti_Tag = WA_CustomScreen;
tagitem[0].ti_Data = screen;      /* Open on my own screen */
tagitem[1].ti_Tag = WA_Title;
tagitem[1].ti_Data = "RKM Test Window";
tagitem[2].ti_Tag = TAG_DONE;     /* Marks the end of the tag array. */

/* Use defaults for everything else. Will open as big as the screen. */
/* Because all window parameters are specified using tags, we don't */
/* need a NewWindow structure */

if (window = OpenWindowTagList(NULL, tagitem))
{
    /* rest of code */
    CloseWindow(window);
}
```

Notice that window parameters need not be explicitly specified. Functions that utilize tags have reasonable defaults to fall back on in case no valid attribute/value pair was supplied for a particular parameter. This fall back capability is a useful feature. An application only has to specify the attributes that differ from the default, rather than unnecessarily listing all the possible attributes.

The `amiga.lib` support library offers another way to pass TagItems to a function. Rather than passing a tag list, the function `OpenWindowTags()` receives the attribute/value pairs in the argument list, much like `printf()` receives its arguments. Any number of attribute/value pairs can be specified. This type of argument passing is called `VarArgs`. The following code fragment illustrates the usage of `OpenWindowTags()`.

```
struct Window *window;
```

```

/* Just pass NULL to show we aren't using a NewWindow */
window = OpenWindowTags( NULL,
                        WA_CustomScreen, screen,
                        WA_Title, "RKM Test Window",
                        TAG_DONE );

```

Tags are not exclusively for use with the operating system; the programmer can implement them as well. The run-time utility library contains several functions to make using tags easier.

1.5 37 / Tags / Advanced Tag Usage

The previous section provided the background material necessary to start using tags. This section will show how to use the more advanced features of tags using functions within utility library.

Creating a New Tag List	Sequential Access of Tag Lists
Copying an Existing Tag List	Random Access of Tag Lists
Filtering an Existing Tag List	Obtaining Boolean Values
Locating an Attribute	Mapping Tag Attributes

1.6 37 // Advanced Tag Usage / Creating a New Tag List

The `AllocateTagItems()` function can be used to create a new tag array ready for use. The tag array should be passed to `FreeTagItems()` when the application is done with it.

```

struct TagItem *tags;
ULONG tags_needed;

/* Indicate how many tags we need */
tags_needed = 10;

/* Allocate a tag array */
if (tags = AllocateTagItems(tags_needed))
{
    /* ...do something with the array... */

    /* Free the array when your done with it */
    FreeTagItems (tags);
}

```

1.7 37 // Advanced Tag Usage / Copying an Existing Tag List

The `CloneTagItems()` function is used to copy an existing tag array into a new tag array.

```

struct TagItem *otags;    /* Original tag array */
struct TagItem *ntags;    /* New tag array */

```

```

/* Make sure there is a TagItem array */
if (otags)
{
    /* Copy the original tags into a new tag array */
    if (ntags = CloneTagItems(otags))
    {
        /* ...do something with the array... */

        /* Free the array when your done with it */
        FreeTagItems (ntags);
    }
}

```

This function can also be used to implement a function that will insert tag items into an array.

```

struct TagItem *otags;      /* Original tag array */
struct TagItem *tags;      /* New tag array */

/* Insert a couple of tags into an existing tag array */
if (tags = MakeNewTagList (GA_LeftEdge, 10,
                          GA_TopEdge, 20,
                          TAG_MORE, otags))
{
    /* ...do something with the array... */

    /* Free the array when your done with it */
    FreeTagItems (tags);
}

/* This function will create a tag array from tag pairs placed on
 * the stack */
struct TagItem *MakeNewTagList (ULONG data,...)
{
    struct TagItem *tags = (struct TagItem *) &data;

    return (CloneTagItems (tags));
}

```

1.8 37 // Advanced Tag Usage / Filtering an Existing Tag List

Sometimes it is necessary to only allow certain attributes to be visible in a tag list. In order to achieve this, the tag array would need to be filtered.

A number of functions are provided for filtering items in a tag array. They are `FilterTagChanges()`, `FilterTagItems()` and `RefreshTagItemClones()`.

```

/* We want the text entry gadget to receive the following tags */
Tag string_attrs[] =
{
    STRINGA_MaxChars,
    STRINGA_Buffer,

```

```

        STRINGA_TextVal,
        TAG_END,
};

/* These are attributes that the model understands */
Tag model_attrs[] =
{
    CGTA_Total,
    CGTA_Visible,
    CGTA_Top,
    ICA_TARGET,
    ICA_MAP,
    TAG_END,
};

struct TagItem *otags;      /* Original tag list */
struct TagItem *ntags;     /* New, work, tag list */

/* Make a copy of the original for us to work with */
ntags = CloneTagItems (otags);

/* Create a tag list that only contains attributes that are
 * listed in the model_attrs list. */
if (FilterTagItems (ntags, model_attrs, TAGFILTER_AND))
{
    /* Work with filtered tag list (ntags) */

    /* Restore the tag list */
    RefreshTagItemClones (ntags, otags);

    /* Create a tag list that only contains attributes that
     * aren't in the model_attrs list. */
    if (FilterTagItems (ntags, model_attrs, TAGFILTER_NOT))
    {
        /* Work with filtered tag list (ntags) */
    }

    /* Restore the tag list */
    RefreshTagItemClones (ntags, otags);

    /* Create a tag list that only contains attributes that
     * are in the string_attrs list. */
    if (FilterTagItems (ntags, string_attrs, TAGFILTER_AND))
    {
        /* Work with filtered tag list (ntags) */
    }
}

/* Free work tag list. */
FreeTagItems (ntags);

```

1.9 37 // Advanced Tag Usage / Locating an Attribute

To see if an attribute is in a tag array, the TagInArray() function is used.

```

/* See if the listview labels attribute is located in a tag array */
if (TagItemArray(GTLV_Labels, tags))
{
    /* Yes, the attribute is in the list */
}
else
{
    /* No, the attribute isn't in the list */
}

```

The FindTagItem() function will return a pointer to the actual tag that has the desired attribute. This allows you to manipulate the tag or to determine if the attribute exists but just has a NULL value.

```

struct TagItem *tag;

/* See if they are trying to set a sound */
if (tag = FindTagItem(MGA_Sound, attrs))
{
    /* Set the sound attribute to point to the specified sound data */
    tag->ti_Data = sound;
}

```

1.10 37 // Advanced Tag Usage / Sequential Access of Tag Lists

In order to sequentially access the members of a tag array, the NextTagItem() function is used.

```

struct TagItem *tags = msg->ops_AttrList;
struct TagItem *tstate;
struct TagItem *tag;
ULONG tidata;

/* Start at the beginning */
tstate = tags;

/* Step through the tag list while there are still items in the
 * list */
while (tag = NextTagItem (&tstate))
{
    /* Cache the data for the current element */
    tidata = tag->ti_Data;

    /* Handle each attribute that we understand */
    switch (tag->ti_Tag)
    {
        /* Put a case statement here for each attribute that your
         * function understands */
        case PGA_Freedom:
            lod->lod_Flags |= tidata;
            break;

        case GTLV_Labels:
            lod->lod_List = (struct List *) tidata;

```

```

        break;

        /* We don't understand this attribute */
        default:
            break;
    }
}

```

1.11 37 // Advanced Tag Usage / Random Access of Tag Lists

The `GetTagData()` function will return the data for the specified attribute. If there isn't a tag that matches, then the default value is returned.

```

APTR sound;

/* Get the sound data that our function will use. */
sound = (APTR) GetTagData (MGA_Sound, (ULONG) DefaultSound, attrs);

```

1.12 37 // Advanced Tag Usage / Obtaining Boolean Values

Often times data is best represented as simple boolean (TRUE or FALSE) values. The `PackBoolTags()` function provides an easy method for converting a tag list to bit fields.

```

/* These are the attributes that we understand, with the
 * corresponding flag value. */
struct TagItem activation_bools[] =
{
    /* Attribute          Flags */
    {GA_ENDGADGET,        ENDGADGET},
    {GA_IMMEDIATE,        GADGIMMEDIATE},
    {GA_RELVERIFY,        RELVERIFY},
    {GA_FOLLOWMOUSE,      FOLLOWMOUSE},
    {GA_RIGHTBORDER,      RIGHTBORDER},
    {GA_LEFTBORDER,       LEFTBORDER},
    {GA_TOPBORDER,        TOPBORDER},
    {GA_BOTTOMBORDER,     BOTTOMBORDER},
    {GA_TOGGLESELECT,     TOGGLESELECT},

    /* Terminate the array */
    {TAG_END}
};

/* Set the activation field, based on the attributes passed */
g->Activation = PackBoolTags(g->Activation, tags, activation_bools);

```

1.13 37 // Advanced Tag Usage / Mapping Tag Attributes

To translate all occurrences of an attribute to another attribute, the `MapTags()` function is used.

For Release 2, the third parameter of this function is always `TRUE` (tags remain in the array even if they can't be mapped).

```
struct TagItem map_list[] =
{
    /* Original      New */
    {MGA_LeftEdge,   GA_LeftEdge},
    {MGA_TopEdge,    GA_TopEdge},
    {MGA_Width,      GA_Width},
    {MGA_Height,     GA_Height},

    /* Terminate the array */
    {TAG_END},
}

/* Map the tags to the new attributes, keeping all attributes that
 * aren't included in the mapping array */
MapTags(tags, map_list, TRUE);
```

1.14 37 Utility Library / Callback Hooks

The callback features of Release 2 provide a standard means for applications to extend the functionality of libraries, devices, and applications. This standard makes it easy for the operating system to use custom modules from different high level programming languages as part of the operating system. For example, the layers library, which takes care of treating a display as a series of layered regions, allows an application to attach a pattern function to a display layer. Instead of filling in the background of a layer with the background color, the layers library calls the custom pattern function which fills in the layer display with a custom background pattern.

Callback Hook Structure and Function

1.15 37 / Callback Hooks / Callback Hook Structure and Function

An application passes a custom function in the form of a callback Hook (from `<utility/hooks.h>`):

```
struct Hook
{
    struct MinNode h_MinNode;
    ULONG (*h_Entry)(); /* stub function entry point */
    ULONG (*h_SubEntry)(); /* the custom function entry point */
    VOID *h_Data; /* owner specific */
};
```

`h_MinNode`

This field is reserved for use by the module that will call the Hook.

h_Entry

This is the address of the Hook stub. When the OS calls a callback function, it puts parameters for the callback function in CPU registers A0, A1, and A2. This makes it tough for higher level language programmers to use a callback function because most higher level languages don't have a way to manipulate CPU registers directly. The solution is a stub function which first copies the parameters from the CPU registers to a place where a high level language function can get to them. The stub function then calls the callback function. Typically, the stub pushes the registers onto the stack in a specific order and the callback function pops them off the stack.

h_SubEntry

This is the address of the actual callback function that the application has defined. The stub calls this function.

h_Data

This field is for the application to use. It could point to a global storage structure that the callback function utilizes.

There is only one function defined in utility library that relates to callback hooks:

```
ULONG CallHookPkt(struct Hook *hook, VOID *object, VOID *paramPkt);
```

This function calls a standard callback Hook function.

Simple Callback Hook Usage

1.16 37 // Hook Structure And Function / Simple Callback Hook Usage

A Hook function must accept the following three parameters in these specific registers:

- A0 - Pointer to the Hook structure.
- A2 - Pointer to an object to manipulate. The object is context specific.
- A1 - Pointer to a message packet. This is also context specific.

For a callback function written in C, the parameters should appear in this order:

```
myCallbackFunction(Pointer to Hook (A0),  
                   Pointer to Object (A2),  
                   Pointer to message (A1));
```

This is because the standard C stub pushes the parameters onto the stack in the following order: A1, A2, A0. The following assembly language routine is a callback stub for C:

```

INCLUDE 'exec/types.i'
INCLUDE 'utility/hooks.i'

xdef      _hookEntry

_hookEntry:
    move.l  a1,-(sp)           ; push message packet pointer
    move.l  a2,-(sp)           ; push object pointer
    move.l  a0,-(sp)           ; push hook pointer
    move.l  h_SubEntry(a0),a0   ; fetch actual Hook entry point ...
    jsr     (a0)               ; and call it
    lea     12(sp),sp          ; fix stack
    rts

```

If your C compiler supports registerized parameters, your callback functions can get the parameters directly from the CPU registers instead of having to use a stub to push them on the stack. The following C language routine uses registerized parameters to put parameters in the right registers. This routine requires a C compiler that supports registerized parameters.

```

#include <exec/types.h>
#include <utility/hooks.h>

#define     ASM      __asm
#define     REG(x)   register __ ## x

/* This function converts register-parameter hook calling
 * convention into standard C conventions. It requires a C
 * compiler that supports registerized parameters, such as
 * SAS/C 5.xx or greater.
 */
ULONG ASM
hookEntry(REG(a0) struct Hook *h, REG(a2) VOID *o, REG(a1) VOID *msg)
{
    return ((*h->h_SubEntry)(h, o, msg));
}

```

A callback function is executed on the context of the module that invoked it. This usually means that callback functions cannot call functions that need to look at environment specific data. For example, `printf()` needs to look at the current process's input and output stream. Entities like `Intuition` have no input and output stream. This also means that in order for the function to access any of its global data, it needs to make sure the CPU can find the function's data segment. It does this by forcing the function to load the offset for the program's data segment into CPU register `A4`. See your compiler documentation for details.

The following is a simple function that can be used in a callback hook.

```

ULONG MyFunction (struct Hook *h, VOID *o, VOID *msg)
{
    /* A SASC and Manx function that obtains access to the global
     data segment */
    geta4();
}

```

```

    /* Debugging function to send a string to the serial port */
    KPrintf("Inside MyFunction()\n");

    return (1);
}

```

The next step is to initialize the Hook for use. This basically means that the fields of the Hook structure must be filled with appropriate values.

The following simple function initializes a Hook structure.

```

/* This simple function is used to initialize a Hook */
VOID InitHook (struct Hook *h, ULONG (*func)(), VOID *data)
{
    /* Make sure a pointer was passed */
    if (h)
    {
        /* Fill in the hook fields */
        h->h_Entry = (ULONG (*)()) hookEntry;
        h->h_SubEntry = func;
        h->h_Data = data;
    }
}

```

The following is a simple example of a callback hook function.

hooks1.c

1.17 37 Utility Library / 32-bit Integer Math Functions

Utility library contains some high-speed math functions for 32-bit integer division and multiplication. These functions will take advantage of available processor instructions (like DIVUL), if a 68020 processor or higher is present. If not, these functions will mimic those instructions in 68000 only instructions, thus providing processor independency.

Currently the following functions are implemented:

	SDivMod32()	Signed 32 by 32-bit division and modulus.
	SMult32()	Signed 32 by 32-bit multiplication.
	UDivMod32()	Unsigned 32 by 32-bit division and modulus.
	UMult32()	Unsigned 32 by 32-bit multiplication.

Table 37-3: Utility Library 32-bit Math Functions

The division functions return the quotient in D0 and the remainder in D1. To obtain the remainder in a higher level language, either a compiler specific instruction to fetch the contents of a specific register must be used (like `getreg()` in SAS C) or a small assembler stub.

Following a simple example of the usage of the 32-bit integer math functions in C.

```
uptime.c
```

1.18 37 Utility Library / International String Functions

The international string functions in utility library are a way to make use of a future localization library which Amiga, Inc. will provide. When the localization library is opened, the functions will be replaced by ones which will take the locale as defined by the user into account. This means that the compare order may change according to the locale, so care should be taken not to rely on obtaining specific compare sequences.

Currently implemented are:

Stricmp()	Compare string case-insensitive.
Strnicmp()	Compare string case-insensitive, with a specified
	length.
ToLower()	Convert a character to lower case.
ToUpper()	Convert a character to upper case.

Table 37-4: Utility Library International String Functions

These functions operate in the same manner as their ANSI C equivalents, for the most part. For more information, see the "Utility Library" Autodocs in the Amiga ROM Kernel Reference Manual: Includes and Autodocs. Here is a simple example of the usage of the international string functions.

```
istr.c
```

1.19 37 Utility Library / Date Functions

To ease date-related calculations, the utility library has some functions to convert a date, specified in a ClockData structure, in the number of seconds since 00:00:00 01-Jan-78 and vice versa. To indicate the date, the ClockData structure (in <utility/date.h>) is used.

```
struct ClockData
{
    UWORD sec;      /* seconds (0 - 59) */
    UWORD min;      /* minutes (0 - 59) */
    UWORD hour;     /* hour (0 - 23) */
    UWORD mday;     /* day of the month (1 - 31) */
    UWORD month;    /* month of the year (1 - 12) */
    UWORD year;     /* 1978 - */
    UWORD wday;     /* day of the week (0 - 6, where 0 is Sunday) */
}
```


Function	Description
AllocateTagItems()	Allocate a TagItem array (or chain).
FreeTagItems()	Frees allocated TagItem lists.
CloneTagItems()	Copies a TagItem list.
RefreshTagItemClones()	Rejuvenates a clone from the original.
FindTagItem()	Scans TagItem list for a tag.
GetTagData()	Obtain data corresponding to tag.
NextTagItem()	Iterate TagItem lists.
TagInArray()	Check if a tag value appears in a Tag array.
FilterTagChanges()	Eliminate TagItems which specify no change.
FilterTagItems()	Remove selected items from a TagItem list.
MapTags()	Convert ti_Tag values in a list via map
PackBoolTags()	Builds a "Flag" word from a TagItem list.

1.22 37 / Function Reference / Callback Hook Function Reference

The following are brief descriptions of the utility library functions which pertain to callback hooks.

Table 37-7: Utility Hook Functions

Function	Description
CallHookPkt()	Call a standard callback Hook function.

1.23 37 / Function Reference / 32-Bit Integer Math Function Reference

The following are brief descriptions of the utility library functions which pertain to 32-bit integer math.

Table 37-8: Utility 32-Bit Math Functions

Function	Description
SDivMod32()	Signed 32 by 32-bit division and modulus.
SMult32()	Signed 32 by 32-bit multiplication.
UDivMod32()	Unsigned 32 by 32-bit division modulus.
UMult32()	Unsigned 32 by 32-bit multiplication.

1.24 37 / Function Reference / International String Function Reference

The following are brief descriptions of the utility library functions which pertain to string operations using the international ASCII character set.

Table 37-9: Utility International String Functions

Function	Description
Stricmp()	Compare strings, case-insensitive.
Strnicmp()	Compare strings, case-insensitive, with specified length.
ToLower()	Convert a character to lower case.
ToUpper()	Convert a character to upper case.

1.25 37 / Function Reference / Date Function Reference

The following are brief descriptions of the utility library functions which pertain to date conversion.

Table 37-10: Utility Date Functions

Function	Description
CheckDate()	Check the legality of a date.
Amiga2Date()	Calculate the date from a specified timestamp.
Date2Amiga()	Calculate the timestamp from a specified date.