@c -*-texinfo-*-

Triton

Release 1.4

Developer Documentation

1 Introduction

1.1 Overview

The Triton GUI creation system offers an easy way to create good looking GUIs for your applications. It can be used with a variety of programming languages. Currently supported are the following languages and development systems:

- C (SAS/C 6.50+, GCC (tested with 2.5.8 and 2.6.0), DICE)
- Oberon (AmigaOberon)
- Modula-2 (M2Amiga)
- E (AmigaE)
- Assembler
- Basic (BlitzBasic)
- Pascal (KickPascal, MaxonPascal)

  Note that the Pascal interfaces are not yet included in this release of the Triton developer package. Please contact the author (Sotirios Pappas <sotto@trkpool.rhein-ruhr.de>) directly.

(If you create a Triton support system for any other language or development system and are willing to maintain it for future updates, please contact me for inclusion into the main Triton Developer distribution.)

This document describes how to use Triton in your own applications by providing you with some overview topics and a step by step introduction. It does **not** describe in detail the functions, structures, tags, etc. which are used, so you should always look them up in the autodocs file (see Section 1.4 [Introduction - Autodocs], page 3) and the C header file '`Developer/Include/libraries/triton.h`'.

The examples in this document assume that you're using the C support files with the SAS/C compiler. There may be differences in other languages or even with other C development systems.

1.2 OOP Internals

Although Triton offers a mostly procedural API, it is based on an object-oriented system. As a Triton user you will never see Triton objects directly, but instead reference them through IDs.

1.3 Class Tree

The following classes are available in Triton:

```
Object                  Abstract root class
‘-- DisplayObject       Abstract class for window contents
    +-- Button          BOOPSI button gadget
    +-- CheckBox        GadTools CheckBox
    +-- Cycle           GadTools Cycle and MX gadget
    +-- DropBox         AppIcon dropping box
    +-- FrameBox        Framing or grouping box
    +-- Group           Triton’s layout engine
    +-- Image           Image
    +-- Line            3D line
    +-- Listview        GadTools Listview
    +-- Palette         GadTools Palette gadget
    +-- Progress        Progress indicator
    +-- Scroller        GadTools Scroller
    +-- Slider          GadTools Slider
    +-- Space           Empty space
    +-- String          GadTools String gadget
    ‘-- Text            Text
```

Descriptions for all classes can be found in the Triton autodocs (see Section 1.4 [Introduction - Autodocs], page 3) under `triton.library/class_<classname>`.

1.4 Autodocs

Documentation on all Triton classes and ‘`triton.library`’ functions can be found in the Triton autodocs file ‘`Developer/Autodocs/triton.doc`’. The function descriptions follow the usual syntax as used in the AmigaOS autodocs. The Triton classes are listed with the prefix `triton.library/class_` and are described using the following keywords:

- `NAME`

  The name of the class and a short description.

- `SUPERCLASS`

  The superclass. If the comment `(no attributes inherited)` is added, all attributes (see below) are described in this autodoc clip. Otherwise all attributes of the superclass are inherited.

- `SYNOPSIS`

  The tags which invoke the creation of an instance of the class.

- `ATTRIBUTES`

The attributes which are created for every instance of the class. All attributes are listed with their tag names, except for the default attribute which is listed as `<Default>`. See above for superclass attributes.

- `MESSAGES`

  This section describes all messages which objects of this class are currently able to send. If no messages are described, this doesn't mean that none are sent. Not all class descriptions have this section yet.

2 Tutorial

2.1 Applications

In order to use '`triton.library`' you must first of all open it, like any other shared library, as described in the `Amiga ROM Kernel Reference Manual`, volume `Libraries`. Before quitting, you have to close it again. Since release 1.1 '`triton.library`' is a single-base library, so you can share one instance of '`triton.library`' between several tasks. The following code makes sure that you can use functions from '`triton.library`' release 1.2 or higher:

```
#include <libraries/triton.h>
#include <clib/triton_protos.h>
#include <pragmas/triton_pragmas.h>

struct Library *TritonBase;

int main(void)
{
  if(TritonBase=OpenLibrary(TRITONNAME,TRITON12VERSION))
  {
    /* Use functions from triton.library... */
    CloseLibrary(TritonBase);
  }
  else
  {
    /* React on the error... */
  }
  return 0;
}
```

All Triton programs are based on a Triton application structure (struct TR_App). This structure is the connection which keeps all Triton parts of your application together. Before using any other Triton functions, you have to create a Triton application, and you must delete it again before your program quits (and before you close '`triton.library`' of course). Any Triton application must have at least a short name, which is used by Triton to identify the application. All other tags are optional. A typical code segment could look like this:

```
struct TR_App *myApp;
```

```
if(myApp=TR_CreateAppTags(
  TRCA_Name,    "MyApp",
  TRCA_Release, "1.0",
  TRCA_Version, "42.113",
  TRCA_Date,    "3.11.94",
  TAG_END))
{
  /* Use myApp for other Triton functions... */
  TR_DeleteApp(myApp);
}
else
{
  /* React on the error... */
}
```

The linker library '`triton.lib`' offers an easier way to open '`triton.library`' and create a Triton application structure. The two examples from above can be combined into the following short version:

```
#include <libraries/triton.h>
#include <proto/triton.h>

int main(void)
{
  if(TR_OpenTriton(TRITON12VERSION,
    TRCA_Name,    "MyApp",
    TRCA_Release, "1.0",
    TRCA_Version, "42.113",
    TRCA_Date,    "3.11.94",
    TAG_END))
  {
    /* The opened application is called 'Application' */
    TR_CloseTriton();
  }
  else
  {
    /* React on the error... */
  }
  return 0;
}
```

2.2 Projects

A Triton Project is the next smaller entity of a Triton GUI. Currently a project contains exactly one Intuition window, but in future versions of Triton it could be possible to attach more windows to a project. Upon opening a project, you have to specify all objects which are to be displayed in the window, the window's menu and some tags describing window properties.

At first you have to specify the window properties, then the menus and finally one (!) object. Normally this object will be a group which contains other objects or groups.

Note that all tags are optional. You may just as well open a window without any tags. This will result in a small window, consisting only of the close and depth gadget and a small dragging bar, being opened on the default public screen's title bar.

As with naming Triton applications, you should give each project at least a unique ID which is used by Triton e.g. for remembering the window dimensions.

Opening and closing a project without any objects and menus could look like this:

```
struct TagItem dummyTags=
{
  TRWI_Title, (ULONG) "A dummy window",
  TRWI_ID,            42,
  TAG_END
};

void dummyFunction(void)
{
  struct TR_Project *dummyProject;

  if(dummyProject=TR_OpenProject(Application,dummyTags))
  {
    /* Opened successfully */
    TR_CloseProject(dummyProject);
  }
  else
  {
    /* React on the error... */
  }
}
```

Of course this way of opening a project via a static TagItem list doesn't allow to insert object parameters or localized strings. Since you need these features most times, you should instead use a dynamical TagList which is built on the stack at runtime. Be sure to set a large enough stack! Some development systems offer an automatic stack setting in their startup code, which comes handy in those situations. Please do not rely on the user to set the stack!

And now back to our code, this time with a dynamic list. Imagine a function `STRPTR GetLocStr(int num)` which gives you a localized string.

```
void dummyFunction(void)
{
  struct TR_Project *dummyProject;
```

```
  if(dummyProject=TR_OpenProjectTags(Application,
    TRWI_Title, (ULONG) GetLocStr(MSG_WINTITLE_DUMMY),
    TRWI_ID,            42,
    TAG_END))
  {
    /* Opened successfully */
    TR_CloseProject(dummyProject);
  }
  else
  {
    /* React on the error... */
  }
}
```

The ID 42 in the above examples has been chosen randomly. This is no problem as long as you stick to this ID in future updates of your application. Otherwise the window dimensions would be wrong and the user would have to adjust them again.

In order to not use IDs twice, it can be useful to enumerate them:

```
enum windowIDs {WINID_DUMMY=1, WINID_FOO, WINID_BAR};
```

Note that window IDs must be different from 0. 0 is equal to no ID at all.

See the autodoc clip `triton.library/TR_OpenProject()` for details about the supported tags.

2.3 Menus

Any project definiton may contain menus. The menu tags must follow the window tags immediately, before the object tags. If a menu has got an object ID, the TRAT_ID tag must be the last one in the menu definition. Object IDs are more than Project IDs. Project IDs are used only internally by Triton. As an application programmer, you have to reference Projects by a pointer to their `struct TR_Project`. Objects instead are not referenced through pointers, but only through object IDs. The same is true for menus, which use ordinary object IDs.

A typical menu defintion could look like this:

```
TR_OpenProjectTags(...
  TRMN_Title,   (ULONG) "Project",
    TRMN_Item,  (ULONG) "?_About",       TRAT_ID, 1,
    TRMN_Item,  (ULONG) TRMN_BARLABEL,   TRAT_ID, 2,
    TRMN_Item,  (ULONG) "Q_Quit",        TRAT_ID, 3,
  ...);
```

The title of a menu item or sub-item can be `TRMS_BARLABEL` to insert a standard separator bar. Keyboard shortcuts can be specified by beginning the menu label with the shortcut key followed by an underscore character and the real menu label. These simple shortcuts (with the right Amiga key) are processed automatically. You can create a so-called 'extended shortcut' by starting a label with an underscore character, then the text for the shortcut, another underscore and the menu label. These extended shortcuts are only displayed with AmigaOS 3.0 or higher and they are not managed automatically. You have to listen to incoming keyboard events in order to handle these shortcuts yourself.

See the `Menus` window of the Triton demo application for a more detailed example on menus.

2.4 Objects

The heart of every Triton GUI are the objects. Currently only instances of subclasses of the `DisplayObject` class can be created. See Section 1.3 [Introduction - Class Tree], page 3, for a short list of all classes. Detailed descriptions can be found in the autodocs.

Objects are created by inserting an object tag into a project definition. For example, a checked CheckBox object with an ID of 99 can be created the following way:

```
TROB_CheckBox,   NULL,
  TRAT_Value,    (ULONG) TRUE,
  TRAT_ID,       99,
```

All available tags are listed in the autodocs. The `<Default>` tag's value has to be inserted directly in the `TROB_<Class>` tag as its data. All other tags are inserted normally after the initial class tag.

Note: It is useful to `enum`erate object IDs just like project IDs. But in contrast to project IDs, object IDs may be used more than once. If you use an object ID several times within the same project, the objects with that ID will be linked together and whenever an attribute (only selected attributes, notably `TRAT_Value`!) of one object changes, this change will be broadcast to all other objects with that ID as if you had notified them yourself with `TR_SetAttribute()`. This is particularly useful to link a CheckBox gadget and a checkable menu item together.

See the `Connections` window of the Triton demo application for a more detailed example on attribute broadcasting.

2.5 Layout

One of the most important classes is `Group`. It implements Triton's layout engine.

`Group` objects offer two kinds of directions:

1. Horizontal (primary direction; secondary direction is vertical). Created with `TRGR_Horiz`.
2. Vertical (primary direction; secondary direction is horizontal) Created with `TRGR_Vert`.

3 different layout types are available for each instance of `Group` (adjustable with the group flags). They affect the primary direction only:

1. `TRGR_PROPSHARE`: Divides all objects proportionally to their minimum size. All spaces retain their minimum size and are not resizable. Non-resizable non-space objects do also stick to their minimum sizes (well, they don't really have a choice, do they? ;).
2. `TRGR_EQUALSHARE`: Same as `TRGR_PROPSHARE` except that all non-space objects have the same size. Their minimum size equals the biggest minimum size of the individual objects.
3. `TRGR_PROPSPACES`: All non-space objects retain their minimum sizes all the time and do not get stretched. Instead the spaces are stretched proportionally to their minimum sizes.
4. `TRGR_ARRAY`: This group builds the top group of an array. In order to create an array, you have to set up an outer `TRGR_ARRAY` group (horizontal for a column array, vertical for a line array) and fill it with single objects (notably spaces) or `TRGR_PROPSHARE` groups in the opposite direction. All elements of the inner groups will be aligned to build an array. Inner groups which have the flag `TRGR_INDEP` set will not be aligned. They are mainly used to insert named separator bars (created with `TROB_Line` and `TROB_Text`) into an array.

Currently the only objects which are treated as spaces in the above scheme are instances of class `Space`.

The behaviour of the group in its secondary direction can be changed, too. Two additional flags are available for that purpose. In most cases you may want to set both of them:

1. `TRGR_ALIGN`: All resizable objects (i.e. resizable in the secondary dimension of the group) are strechted to fit the full space occupied by the group.
2. `TRGR_CENTER`: All non-resizable objects are centered in the group. Without this flag they get aligned to the left or top border.

It is also possible to keep a group at its minimum size and don't allow it to be stretched in either or both directions. This can be accomplished with the `TRGR_FIXHORIZ` and `TRGR_FIXVERT` flags.

A group can be created like any other object. It takes other objects as its arguments. Every group has to be terminated with a `TRGR_End` tag. A typical horizontal group, which contains a CheckBox, a space and again a CheckBox, would look like this:

```
TRGR_Horiz,      TRGR_PROPSHARE|TRGR_ALIGN|TRGR_CENTER,

  TROB_CheckBox, 0,
    TRAT_ID,     1,

  TROB_Space,    TRST_NORMAL,

  TROB_CheckBox, 0,
    TRAT_Value,  TRUE,
    TRAT_ID,     2,

  TRGR_End,      0
```

See the `Groups` window of the Triton demo application for a more detailed example on groups and arrays.

2.6 Macros

The C header file '`Developer/Include/libraries/triton.h`' contains quite a lot of macro definitions which make creating a Triton GUI much easier. With the help of the preprocessor `Mac2E` these macros are also available in `AmigaE`. Other languages or compilers may support similar or different macros or none at all. If no macros are available in your development system, you have to do it the traditional way. If you can use macros, use them.

The static taglist definition can be written a bit simpler with macros:

```
ProjectDefinition(dummyTags)
{
  /* Insert project tags here */
};
```

But the main use for macros are the tags themselves. For example, the tags from the dummy window example (see Section 2.2 [Tutorial - Projects], page 6) would look like this:

```
  WindowTitle("A dummy window"),
  WindowID(42),
  EndProject
```

Have a look at the macro definitions and the supplied demo applications for more detailed information about the macros.

2.7 Polling Loop

After creating a GUI you have to handle user input. In order to accomplish this task, Triton offers a polling system which resembles the Intuition IDCMP polling system very much. A basic polling loop looks like this:

```
void handleDummyWindow(void)
{
  BOOL closeMe=FALSE;
  struct TR_Message *trMsg;

  /* Open the window... */

  while(!closeMe)
  {
    TR_Wait(Application,NULL);
    while(trMsg=TR_GetMsg(Application))
    {
      switch(trMsg->trm_Class)
      {
        case TRMS_CLOSEWINDOW:
          closeMe=TRUE;
          break;

        case TRMS_ACTION:
          switch(trmsg->trm_ID)
          {
            case ID_FOO:
              /* Do something... */
              break;
            case ID_BAR:
              /* Do something... */
              break;
          }
          break;

        case TRMS_NEWVALUE:
          switch(trMsg->trm_ID)
          {
            /* New value is in trMsg->trm_Data */

            case ID_MYCHECKBOX:
              /* Do something... */
              break;
            case ID_MYSLIDER:
              /* Do something... */
              break;
          }
          break;
```

```
            case TRMS_ERROR:
              puts(TR_GetErrorString(trMsg->trm_Data));
              break;
          }
          TR_ReplyMsg(trMsg);
        }
      }

      /* Close the window... */
    }
```

Note: Don't forget to reply all messages!

You could add a `switch()` for the project which sent the message if you have more than one project opened, but a better way is to use unique object IDs. This does also yield the advantage that moving an object from one window to another does not require any change in the message handling.

It is currently not very well documented which messages are sent by the different objects. As a general rule, activatable objects send `TRMS_ACTION` and objects which have a `TRAT_Value` or a similar modifiable tag, send `TRMS_NEWVALUE`.

Here is a more detailed description of the message types:

- `TRMS_CLOSEWINDOW`: The user has pressed the close gadget or the `Esc` key (if it hasn't been disabled with `TRWF_NOESCCLOSE`).
- `TRMS_ERROR`: An error occured. `TR_Message.trm_Data` contains the error code. You can use `TR_GetErrorString()` to generate a user-readable error message from this error code.
- `TRMS_NEWVALUE`: An object's attribute has changed. `TR_Message.trm_Data` contains the new value for the attribute.
- `TRMS_ACTION`: An object has been activated somehow (e.g. a button has been pressed or its keyboard shortcut was used instead).
- `TRMS_ICONDROPPED`: An icon has been dropped over a window. You will get this message only if an object in your window reacts on dropped icons or you ask directly for it by specifying the project flag `TRWF_APPWINDOW`. Otherwise Workbench will not allow you to drop icons over a Triton window. `TR_Message.trm_ID` contains the ID of the object over which the icon was dropped or 0 if it was dropped over an object without an ID. `TR_Message.trm_Data` contains a pointer to the `struct AppMessage` which was sent by Workbench.
- `TRMS_KEYPRESSED`: A key has been pressed and Triton was not able to identify it (e.g. as a keyboard shortcut). `TR_Message.trm_Data` contains the ASCII code (if available, otherwise 0), `TR_Message.trm_Code` contains the RawKey code and `TR_Message.trm_Qualifier` the qualifier bits.

- `TRMS_HELP`: The user requests help for the object specified by `TR_Message.trm_ID` or for the whole window if the ID is 0.

A description of all fields of the `TR_Message` structure can be found in the C header file '`Developer/Include/libraries/triton.h`'.

2.8 Messages

Sometimes you may wish to query an object's attribute directly without waiting in a polling loop and then putting it down somewhere else. This is especially useful for string gadgets. You should *not* read their contents when you receive a `TRMS_NEWVALUE` message, because it is possible to exit a string gadget without triggering such a message. Instead you have to read the contents directly before you need them.

Attributes can be read with the `TR_GetAttribute()` function, which will return the value of an object's or project's attribute. Specify the object's ID or 0 for a project attribute. You can get the default attribute of an object by setting `Attribute` to 0. Otherwise set it to an attribute tag.

Setting a new attribute works just the same way with the function `TR_SetAttribute()`. Again you may also modify project attributes. See the autodoc clip for a list of them. After setting a new attribute the on-screen representation of it will be updated if necessary.

If you want to send custom object messages (`TR_SetAttribute()` and `TR_GetAttribute()` invoke the messages `TRDM_SETATTRIBUTE` and `TRDM_GETATTRIBUTE`), you have to use `TR_SendMessage()`. The autodocs will tell you which classes accept which messages.

Note: Do not confuse `Object Messages` (`TROM_...`) which are sent to Triton objects with `Triton messages` (`TRMS_...`) which you receive in your polling loop.

2.9 Help

Triton offers two ways of providing help for the user of a Triton application:

- QuickHelp

  `QuickHelp` creates and handles requester bubbles (actually they are implemented as rectangular boxes at the moment). All you have to do is provide a `TRDO_QuickHelpString` attribute for every object which should have QuickHelp available. Then switch on QuickHelp with `TRWI_QuickHelp` (can be set when opening a project or modified later with

`TR_SetAttribute()`). When the user moves the mouse pointer over an object which has a QuickHelp string attached, a small window will pop up near the object where the mouse pointer is located. This window contains the specified QuickHelp string. As soon as the mouse pointer leaves the borders of the object, the help window will disappear.

- Manual help

  If you set `TRWF_HELP` when opening a project, you will receive a `TRMS_HELP` message whenever the user presses the `Help` key. As usual `TR_Message.trm_ID` contains the ID of the object for which the user requested help or 0 if no object could be assigned to the help request (you should normally provide help for the whole window in that case). The most common way to react on `TRMS_HELP` is to pop up an AmigaGuide document describing the object for which the user requested help. If you do not want to provide more extensive help than you do with the QuickHelp feature (see above), you can also pop up a requester containing the QuickHelp string for the specified object. The main window of the main Triton demo application shows how to do that.

2.10 Requesters

Triton offers two requester functions:

- `TR_EasyRequest()` creates and handles a simple requester with some lines of text (different text styles are possible) and a row of buttons at the lower end. You should use this function whenever possible. Not only is it easier to use than `TR_AutoRequest()`, but it could be made user-configurable in future Triton releases.
- `TR_AutoRequest()` is a simple message polling loop which opens a Triton window, waits until a `TRMS_ACTION` message comes in and returns the ID of the object which triggered the message. This function is often used in combination with the requester macros (`BeginRequester()`, `BeginRequesterGads` and `EndRequester`).

Please consult the autodocs for more detailed information about these functions. The main demo application contains examples for both types of requesters.

If you need a more sophisticated requester (e.g. a string requester), you have to use your own message polling loop (see Section 2.7 [Tutorial - Polling Loop], page 12). You may still use the requester macros though.

3 Style Guide

3.1 Guidelines

Please keep the following general rules in mind when designing and implementing a GUI with Triton:

- Window size

  Please make sure that all windows of your GUI fit on a standard `PAL Hires` screen (640*256) with '`topaz.font`' in size 8.

- Keyboard shortcuts

  Please add keyboard shortcuts to as many gadgets and menus as possible. It is very annoying for a user who works without a mouse most of the time to grab his mouse just because you forgot to add keyboard shortcuts.

- Test your GUI

  You should always test your GUI with different font sizes (especially fixed-width and proportional fonts with different heights) and a non-standard window background to avoid making assumptions about the layout and coloring of your GUI resulting from observed behaviour, which is quite often wrong. Take special care not to use text spaces for spacing in your GUI (e.g. in array labels).

- Stack size

  If your application requires a stack size above 4096 bytes (e.g. because of a huge TagList with a Triton project definition, which is created on the stack during runtime), make sure that the stack does indeed have that size. Some development systems offer an automatic stack setting in their startup code, which comes handy in those situations. Please do not rely on the user to set the stack!

- And finally:

  Don't forget the `Amiga User Interface Style Guide`...

4 Odds & Ends

4.1 FAQ

Here are some frequesntly asked questions and their answers:

1. Q: Why shouldn't I use a simple `TR_Wait()` for dummy windows?

   A: Triton has to process inputs from Intuition. This is done in `TR_GetMsg()`. But of course you have to call this functions so that Triton can process its input. After `TR_Wait()` returns, you have to run though a `while(TR_GetMsg())` loop as usual. An easier way to create a dummy window is using `TR_AutoRequest()`.

2. Q: How do I handle a progress indicator?

   A: As already explained in answer 1, you have to call `TR_GetMsg()` regularly. When a progress indicator is displayed, you should update it regularly and run through a `while(TR_GetMsg())` loop every time. See the supplied Progress Indicator demo application for details.

3. Q: How do I activate a string gadget?

   A: Beginning with Triton 1.3 (V4) you can use `TR_SendMessage(Project,ID,TROM_ACTIVATE,NULL)`.

Index

Table of Contents