RMP
A multimedia platform for the ROME operating system

May 1, 2001

Wolfgang Reißnegger
Distributed Systems Software Group
C&C Research Laboratories
NEC USA, Inc.
4 Independence Way
Princeton, NJ 08540-6634

You can get the current version of this document at `http://rome.sourceforge.net`
For questions and comments `mailto:rome-admin@lists.sourceforge.net`

Abstract

This technical report describes the *Rome Multimedia Platform* (RMP), a modular, object oriented, extendable development environment for audio and video application programming for the ROME operating system. It shows the structure of the platform, describes the features of its toolkits and how they interact with user applications and the rest of the system.

It also provides information on how to extend the toolkits and how to incorporate new components into the platform by taking advantage of its modular design.

Contents

List of Figures

1 Introduction

1.1 ROME

ROME is an operating system that has been designed to manage high speed data streams within a multimedia environment. The system is highly modular, with functionality split between multiple processes. To ensure a high throughput with minimal overhead ROME provides a *zero copy architecture* where pointer references to data are passed around instead of data being copied. The goal of this approach is to maximize the utilization of a given hardware configuration making it possible to:

- Create high performance systems with much higher data throughput than conventional systems;
- Build fast applications for embedded, mobile systems with lower performance CPU .

Considering the latter point — especially in a world with more and more mobile computing devices, a fast operating system able fully to exploit the available hardware is essential for reasonable performance. ROME is designed to serve this purpose.

1.1.1 First application concepts

Both types of application have already been proven to work well with ROME in demonstration systems: Network-based video delivery systems on the MCCP prototype and VR4300 development boards have shown high bandwidth applications running under ROME, and a 4300-based embedded processor used in a WebPanel, a mobile end user terminal, could easily run a graphical demo.

These demo application ran successfully on ROME over a long time period, proving the stability and reliability of ROME. However, being specifically written for the system hardware configuration, the application code was neither modular nor re-usable.

1.1.2 Problems

Customizing applications for every individual target system's hardware configuration is time consuming and inefficient. Building customized monolithic code might be acceptable for demo applications on development systems, but other, more efficient and easier ways of building user applications had to be found for building flexible end user applications in a reasonable amount of time without reinventing the wheel.

For example, porting a device driver of a graphics chip integrated directly into the application to another system might require substantial rewriting, possibly breaking system code dependencies and making other parts of the code unusable. This could require immense recoding effort throughout the whole code, making timely application development a real impossibility.

1.1.3 New approach for application development

ROME's architecture of shared libraries and fast data-passing paths provides a good foundation for designing and developing a more flexible support platform for multimedia applications. As ROME does not have to be compliant with any existing windowing system or user interfaces, new designs and techniques can be explored. The *Rome Multimedia Platform* (RMP) was designed to exploit those capabilities of ROME. RMP provides:

- Modular design
- Flexibility and extendibility
- Hardware independent re-usable components
- Small, easy to port, hardware-dependent components
- A generic API for user application development
- Maximized utilization of ROME capabilities

Minimizing the amount of hardware dependent components in RMP makes it easy to port RMP to other hardware configurations without rewriting any of the generic components or the user application.

1.1.4 Future development

The RMP design has already been integrated with the other subsystems of ROME, including local file and NFS support. The next step will be to improve configuration capabilities and flexibility using the emerging system wide configuration database and the existing ROME system manager. A scripting language is currently under development for fast prototyping of a variety of user applications.

1.2 RMP

The aim of RMP is to provide an integrated programming environment for multimedia applications executing on a wide variety of hardware configurations. The initial development for RMP was basically driven by potential applications for the WebPanel, and so focuses mainly on graphical and sound components, but RMP's flexible design opens the way for many other components to be embedded into the RMP architecture.

1.2.1 RMP overview

RMP is a collection of modules, implementing processes or shared libraries within ROME, which can be grouped into three layers. Each layer has a well defined task within the system.

As Figure 1 shows, RMP utilizes the full range of ROME communication mechanisms. This design requires that the application will conform to the ROME programming style and provide a message dispatcher to forward messages to the toolkit event or message handler. This requirement is similar to toolkit API's in other architectures where applications have to call a *main loop* function for event processing.

Hardware dependencies in RMP are only located in the *device driver layer*. When porting RMP to another hardware configuration, only affected *device driver modules* have to be changed. All other components in RMP are hardware independent and can be re-used on other hardware platforms without changes.

1.2.2 Modules - RMP processes and shared libraries

The platform contains both processes and shared libraries which collectively comprise the *RMP modules*. The processes use ROME *dataflow messages* to communicate and pass data between them. This avoids unnecessary copies of the data. They also use the ROME *event mechanism* to distribute notifications to upper-level processes.

The shared libraries in RMP have different purposes. Some provide pure library functionality while others are basically an extension for sending messages to lower-layer processes.

Functionality is split between different modules and modules are kept as independent as possible. This design provides several advantages:

- Modules can be added to or removed from the system without changing other code. This is useful if new hardware is added to or removed from the system;
- Modules can be added from or removed from the system during runtime, if the hardware configuration dynamically changes[1];
- Code maintenance on a per module basis is much easier than maintaining a complex system with lots of dependencies;
- Code can be exactly adapted to an existing hardware thus optimizing memory usage and overall performance.

1.3 RMP layering

When creating complex systems, a good design will group different functionalities and tasks within the system. RMP uses *layering* to separate different stages of data processing, introducing three main layers:

- Toolkit layer
- Adapter layer

[1] especially in mobile systems, where removable devices like PC-Cards are common

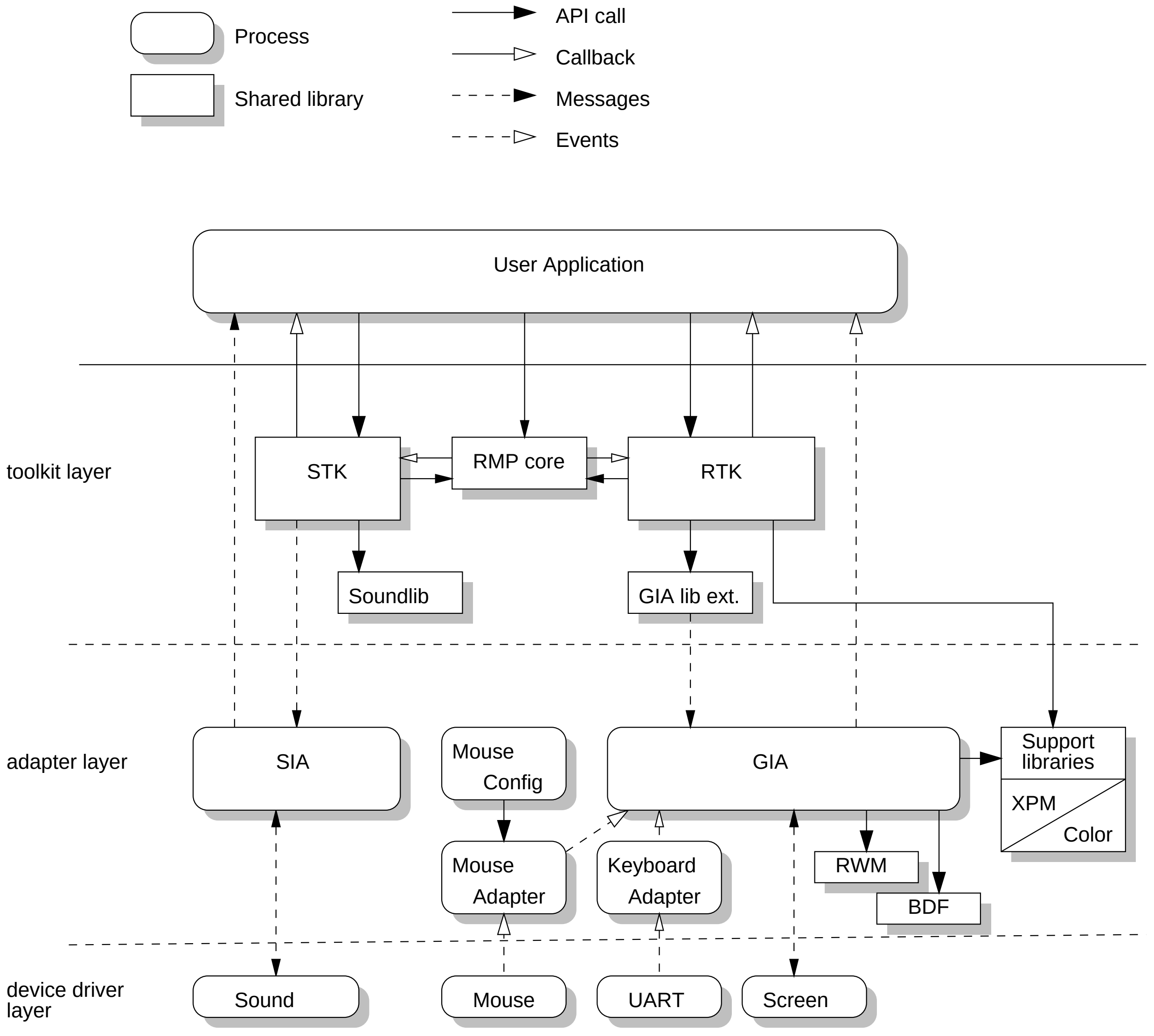


Figure 1: RMP overview

- Device driver layer

These layers separate different degrees of device abstraction within RMP.

1.3.1 Toolkit layer

A *toolkit* is an API for user applications. RMP currently implements two different toolkits. RTK, the *rendering toolkit*, implements windows, buttons, sliders, textfields and other widgets that are necessary to build a graphic-based application. STK, the *sound widget toolkit*, provides widgets for loading, playing and managing sound samples.

As shown in Figure 1, there are no dependencies between the toolkits. Toolkits can independently included or removed from a system or new toolkits can be developed and added to RMP in the future. This, again, shows the flexibility of RMP.

Toolkits are shared libraries. They do not have queues and can not receive messages. They use the processes in the *adapter layer* to communicate with the *device driver processes*. However they implement message or event handlers to handle returned messages from the *adapter layer*. This means that user applications have to call the toolkit event handlers once they receive messages (see section 3.7.2, page 33).

The toolkit APIs are the only interfaces available for user applications. Although it is possible, user applications should rarely access *adapter layer* APIs by themselves.

1.3.2 Adapter layer

The main reason for introducing the *adapter layer* is to manage requests from toolkits, and to prevent resource conflicts if there are parallel requests from different user applications. This is necessary if the hardware (e.g. the sound chip) only allows access for one process or user application at a time. Adapters work as both *multiplexers* for downstream and *demultiplexers* for upstream data, distributing incoming events to the associated process.

All processes that are required to communicate with the *device driver layer* must be located in the *adapter layer*. Usually there exists one adapter process per toolkit or device type. Additional toolkits might simply use already available processes or, depending on the device type, require a new adapter process to be implemented. Figure 1 shows two adapter-layer processes; one for each toolkit. *SIA* is the *Sound Interface Adapter*, and *GIA* is the *Graphics Interface Adapter*.

The rôle of *adapter layer* processes is not necessarily restricted to multi- or de-multiplexing data streams. In the case of GIA a lot of additional functionality is implemented. In addition to GIA's multiplexing and demultiplexing features which take care of distributing mouse or key events to the active target application, GIA provides desktop management, including drawing functions for *drawables* (see section 8.3, page 53) and furthermore provides optional window manager capabilities using a window manager shared library extension (RWM).

1.3.3 Device driver layer

In the *device driver layer* processes control the hardware of the system. For each type of hardware (e.g. a graphics chip) there exists a device driver process . IT has a defined common API which all drivers for this hardware type have to implement.

Device driver processes (or modules) are the *only* processes in RMP that are hardware dependent. When porting RMP to another platform only these modules have to be changed or replaced by modules supporting the new hardware.

1.4 RMP data abstraction

One of the main aspects in layering RMP is the introduction of successive data abstractions. Different components in RMP are kept as independent as possible, thus both requiring other independent data abstractions as well as defining and implementing their own. This mechanism allows higher-level components to be implemented independently from their hardware environment, as they can rely on certain data types, or encapsulations they have to deal with. This transparent mechanism makes RMP components very powerful, re-usable and flexible.

Data abstraction layers are not directly associated with RMP layers, they can be located anywhere in the system. Different data abstraction levels apply to different groups of components, which do not necessarily have to be located in one layer or define a whole layer. Data abstraction in RMP is achieved through following mechanisms:

- Data encapsulation
- Data mapping
- Change of encapsulation (as a subset of data mapping)
- Data translation

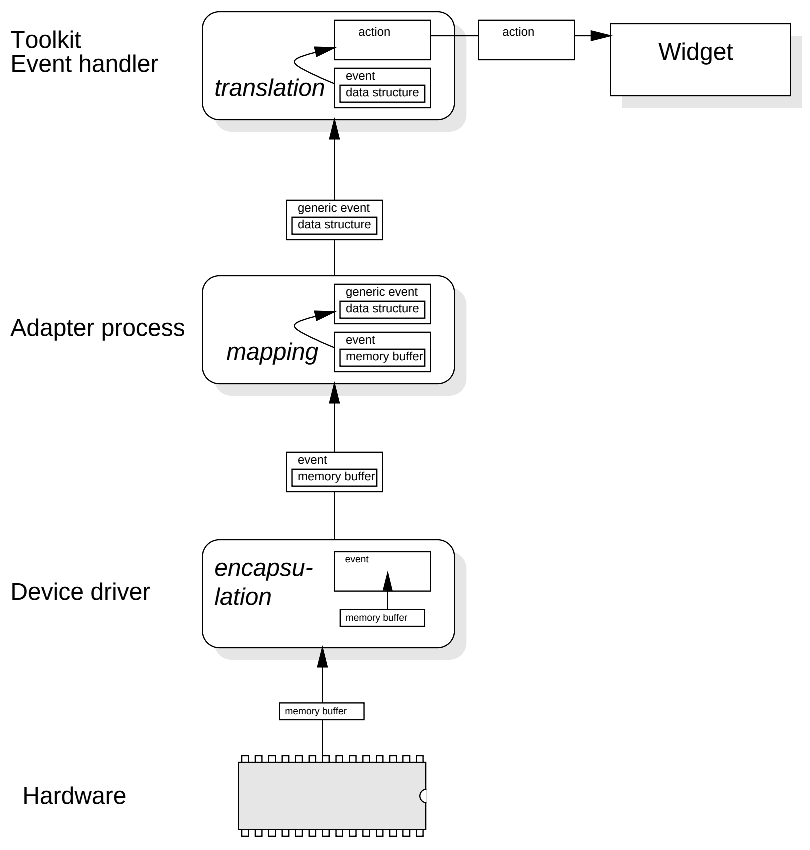


Figure 2: Data abstraction mechanism in RMP

To reach different levels of abstraction, these methods may be used in combination. Figure 2 gives an example of different stages of abstraction.

1.4.1 Data encapsulation

Processes in the device driver layer directly control the hardware of the system. Input and output operation occurs through defined data buffers in system memory or through processor and hardware controller registers. Depending on the hardware type the process may issue DMA operations and has to implement functions for reacting to interrupts that are caused by I/O.

Device driver processes use *messages* or *events* to communicate with the *adapter layer* while *encapsulating* their information into the message or event body. For upstream data, raw memory buffer data will be encapsulated in more abstract structures like `mblks` or events making it also possible to pass this data around in the system. Encapsulation also means providing more detailed information about the data. The device driver will usually include information about the data like the length of the data buffer, or additionally provide information about the data type or even set a timestamp to make synchronization of data streams possible in upper layers of RMP.

Downstream data will be presented encapsulated to the device driver, including additional information that might be necessary for the device driver to interpret the contained data.

The advantage of encapsulating data is that upper layer processes do not have to know anything about the internal representation of the underlying layer. Their functionality is limited to processing the data in the encapsulated form from the lower level and providing information in encapsulated form to the lower level. Changes in single components do not affect any of the other components as long as the encapsulation does not change. Even if a component changes its internal behaviour, this keeps the rest of the system unaffected.

1.4.2 Data mapping

In some cases the structure or type of data changes when they are passed between RMP components. For example, in a device driver process there could be a lot of detailed information available describing an incoming data stream. Usually this kind of detailed low level information and parameters are not required to be passed to another component, because the other components need only know selected parameters of the data stream to process it. In other cases another component simply *must not* know anything about this detailed information. In either case the destination component will use its own data structure, or class of data structure to receive data, which is defined on a per component basis.

When passing data to such a component, *data mapping* is required. The key data values of the source data structure is mapped into the associated fields of the destination data type. Multiple stages of mapping within RMP will result in a data type that represents the original information at an more abstract level.

For example, the *end point* of data mapping within RMP is the abstraction into *widgets* at the user level, where complex data types and behaviour patterns are mapped into a *single, simple data type* for use in applications.

1.4.3 Change of encapsulation

Data transformations can also result in a *change of encapsulation*. Data being encapsulated in a *message* or an *event* can, for example, be mapped into an *action* data structure before being passed to other components. In RMP this mechanism is used to *isolate* different layers of communication from each other to increase the flexibility of the communication mechanism in RMP.

1.4.4 Data translation

Data translation is another key feature of RMP. It provides the ability to generate generic APIs within RMP components. Some components in RMP expect incoming data types and parameters to fit into a specific range, however the source component (or hardware) creates data that does not conform with these requirements. In such cases the data will be *translated* before being passed to the other component. The translation can happen within the source component or in a different component, such as a translation process.

Data translation usually depends on hardware features of the system and is therefore not necessarily a static procedure. This is especially true, if hardware has to be *calibrated* before being used. Translation coefficients can be different on the same hardware due to production tolerances. For these cases, they can be dynamically configured.

2 Widgets

Widgets are the basic components out of which all toolkits in RMP are built. Widgets can be thought of as single building blocks with a well defined functionality, designed to fulfill specific tasks within the toolkit's environment. Together, they implement most of the functionality of a toolkit.

Each individual widget can implement its own functionality and encapsulate private data which is only accessible by the widget itself. Widgets are used through an external API.

2.1 Abstraction

A toolkit can provide a large set of widgets of different internal types. These widgets can have completely different functionality but for the user application they are all represented by one data type. An user application does not have to worry about the details of a certain widget implementation, it simply can reference the widget via *handles* and *names*.

The application uses the widget handles to identify the widget when invoking widget methods for a specific widget type. It can use the *name lookup function* (see section 3.6.2) to find the handle of a widget using its name string. This is like the ROME process names and queue handles.

2.1.1 Handles

User applications need to keep track of widgets they created. This is done by using *widget handles*. Every toolkit function that creates a widget returns a widget handle to the application. The application uses the handle to identify the widget on any further operation. Due to the RMP design, there exists only a *single* handle type used across *all* toolkits (also see section 2.3).

2.1.2 Names

Another way of identifying widgets is through their names, which are assigned on creation of the widgets. Names of nested widgets consist of the names of all their parents, separated by a dot (.) followed by their own name.

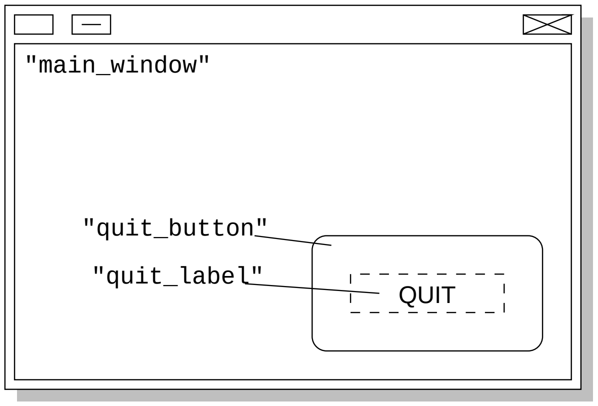


Figure 3: Widget naming

For example, the button label widget in figure 3 would be named

```
"main_window.quit_button.quit_label"
```

Applications are free to choose the names they give to the widgets they create. By using string-based names, it becomes possible to represent them as ROME URLs and so reference widgets across process and system boundaries. The naming scheme enables handles to be specified for each individual widget in a system. This becomes important when sending *actions* to widgets in other processes. For example, RMP *data widgets* can be used for inter process communication through widgets.

To avoid confusion widget names *must not* contain dots and widgets within the same nesting level should be named differently. Widgets with identical names within the same nesting level *can not* be distinguished afterwards. For widgets `""` (no name) is a name, the *empty string*, which is treated the same way as any other name string. If widgets do not have to be referenced by name within the application, they may be given empty names.

2.2 Object-oriented design

Widgets are embedded in the RMP toolkit framework. RMP provides a generic API for user applications to manipulate and manage them. This generic API simplifies application development by abstracting widgets even of different toolkits to a single type and makes application code easier to maintain. To ensure that all widgets work well within this environment they need to have some common properties, namely:

- A set of generic attributes
- A generic API

- A generic behaviour

RMP is designed to provide a single API for managing and manipulating the generic components of widgets across different toolkits. The idea is to be able to apply certain RMP functions to *any* widget type that is or will be implemented in a toolkit. To make this possible, RMP forces an object-oriented widget design by defining the *RMP generic widget class*.

2.2.1 Widget class

When talking about object oriented design, it is necessary to define the terms *class* and *object*. A widget *class* defines a data structure and the behaviour. It is a static, compile-time defined structure that contains attribute fields and function prototypes for the external widget API. At run-time, the class is used to create *instances* of the widget class, also called *widget objects*. There may be many instances of one class.

The RMP generic widget class defines both a data structure and a behaviour. Although it is technically possible to create *instances* of the RMP generic widget class, the RMP API *does not* provide a function to do so; this widget *instance* or widget *object* would provide only very limited functionality and would be rather useless. The purpose of the RMP generic widget class is to provide a base class which toolkits can use to implement new widgets by *inheriting* its features.

2.2.2 Abstract classes

RMP does not provide a function to create widget objects from the RMP generic widget class. This class is only used as a base class to derive sub classes. A class which is only used as a base class is called an *abstract class*.

New toolkits introduce their own *abstract classes*, by deriving them from the RMP generic widget class and adding functionality and attributes to specialize them for the toolkit's needs.

2.2.3 Class hierarchy

Widget classes can be structured to build a class hierarchy. Within the class hierarchy *derived classes* inherit attributes and methods from their *base classes*. When talking about a class relation the inherited class is always called *derived class* while the class from which it is derived is called *base class*. Base classes can also be derived classes of their parent base classes.

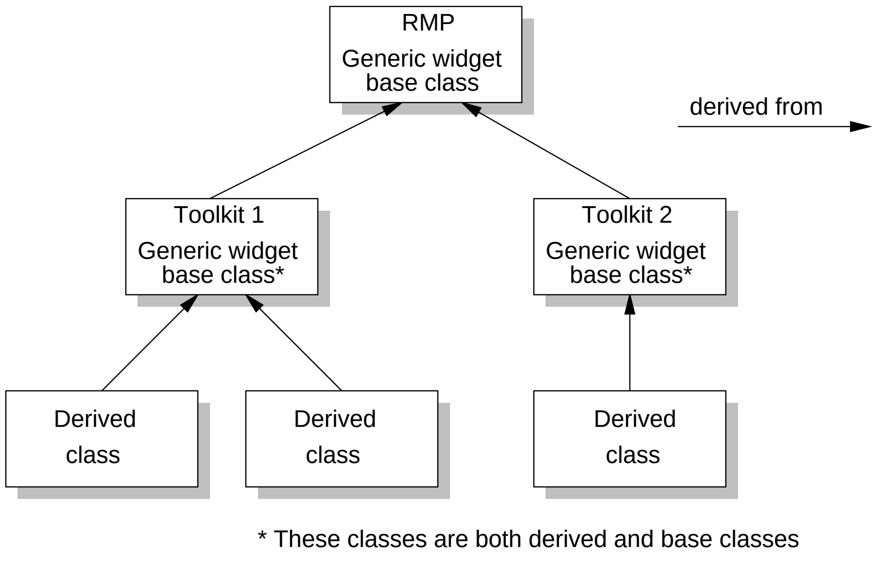

RMP
Generic widget base class
derived from
Toolkit 1
Generic widget base class*
Toolkit 2
Generic widget base class*
Derived class
Derived class
Derived class
* These classes are both derived and base classes

Figure 4: Widget class hierarchy example

Figure 4 shows an example for a widget class hierarchy. The generic base class defines the common denominator for all widgets in the class tree. *Toolkit 1* derives its own generic widget base class from the generic widget base class, so the toolkit 1 generic widget base class is *both* derived class *and* base class depending on the point of view. The same mechanism applies to the *Toolkit 2* generic widget base class.

2.2.4 Widget object

Each *class* defines its own attributes and behaviour. During run-time the application creates instances of classes, called widget *objects*, or just *widgets* for short. These objects have the attributes and features described in the associated class. The application can create many objects of the same class and combine them in any way the RMP API allows it to.

A *class* is only an abstract definition of a widget. Only when a widget *object* is created is memory allocated to accommodate the widget object's attributes. At this point a widget object actually *exists* in the system.

2.2.5 Object structures (nesting)

Widgets of the same or different types can be arranged to build more complex structures. It is possible to nest widgets in order to combine their functionality. When widgets are nested they usually build a widget tree structure. Nesting combines *widget objects* not *widget classes* and must not be confused with class inheritance.

Widgets can also be *connected* via the *action* mechanism (see section 2.8, page 25) which enables them to pass information between each other. Toolkits can use these mechanisms to create more complex widgets. For example, the RTK labeled button creates a structure out of three widgets: a labeled button widget containing a plain button widget and a label widget. For the application only a single *labeled button* widget is immediately visible.

2.3 RMP generic widget class

The *generic widget class* defines the data structure and behaviour of the RMP generic widget. As the RMP generic widget class is the anchor for *all* widgets of *any* toolkit in RMP, any widgets can be referenced as the type of the RMP generic widget.

Within the methods of this class is also the only place where memory for widget objects is allocated or freed during run-time. No other widget class method is allowed to allocate any widget related memory, except for its private data structures.

2.3.1 Internal representation

RMP provides a definition of the RMP generic widget class. This definition *is not* used by user applications, only by toolkit widget implementations and the RMP core. The internal definition of the RMP generic widget class is located in the file:

```
rmp_widget.h
```

and is included by toolkits to obtain the RMP generic widget class properties for deriving their own widgets. The type of the RMP generic widget is defined as:

```
rmp_widget_t
```

which represents the actual C-type of the structure. This definition is only used *internally* by the RMP core implementation. The RMP generic widget class is implemented in the C-module:

```
rmp_widget.c
```

2.3.2 External representation

User applications do not need to know anything about the RMP generic widget class data definitions. They use the toolkit widget class definitions derived from the RMP generic widget class to create more useful widgets.

However, user applications need information about the RMP generic widget class *type*, as all widgets in RMP are referenced through that single type. This information is located in the *external* RMP header file:

```
rmp.h
```

and will be included by user applications. All *external* or *public* definitions of RMP are located in this file. RMP also defines the *RMP generic widget handle* which is defined as:

```
RMP_WIDGET
```

It provides representation of widgets on a very abstract level. Using this handle, the user application can reference *any* widget type in *any* RMP toolkit.

2.4 Generic attributes

The widget common attributes are used to keep data in the widget object that is used by the RMP core to manage single widgets and widget trees. Some of these attributes are set during the widget object *creation*, others are defined and used during the *lifetime* of the widget. As widget trees are dynamic structures, they can be rearranged during run-time. The attributes comprise:

- Name of the widget
- Type of the widget
- A reference to the widget's parent widget
- A references to a child widget
- A reference to a sibling widget
- Generic API callback functions, called *virtual methods*

All widgets *must* be inherited from this class for RMP to work properly.

2.4.1 Name of the widget

The terminal component of the widget name is stored in a data buffer inside the widget object, meaning the name string is copied into the widget name buffer. This means, that there is a maximum length for each component of widget names. If longer names are given to widgets, they are truncated to the maximum length.

2.4.2 Type of the widget

When a widget is created, the type of the widget is stored in the *widget type* field. When methods of widgets are invoked by the user application, the type field is checked to make sure that the widget type is appropriate.

2.4.3 Parent widget reference

Every widget keeps a reference, or *handle*, to its parent widget. The RMP core uses this handle, for example, when sending actions to widgets. This field is set by the RMP core support functions and usually should not have to be changed by widget implementations.

2.4.4 First child widget reference

Here, a handle to the first child in the list of children is stored. This information may be used in many different ways. For example, the *name lookup* function (see section 3.6.2, page 32) uses this field when searching for a widget name within a widget tree.

2.4.5 Sibling widget reference

All children of a widget are kept in a linked list. This field provides the link to the next child of the same parent widget.

2.4.6 Virtual methods

Every widget *must* implement a set of generic methods which are used by the RMP generic core. These are widget *virtual methods*. Each widget registers these methods in the virtual method fields of the widget object. This mechanism allows new widgets either to *inherit* these methods from their parent class or to *override* the virtual methods in order to implement their own functionality.

2.5 Inheritance

Individual toolkit widget classes *inherit* features from the generic widget class. The mechanism of inheritance in RMP is different from the mechanism used, for example, by C++. As RMP is implemented in C it uses another approach to define a class hierarchy.

2.5.1 Data inheritance

The way of inheriting data from another class is relatively simple. Every class is defined by a data structure. Data structures of classes are always defined to the type:

```
XXX_WIDGET_CLASS
```

where *XXX* is the acronym for the toolkit name. For example, the RMP generic widget class type is named:

```
RMP_WIDGET_CLASS
```

The class type is defined in the widget class header file which is included by widget classes which inherit from that type. The header file is also used by user applications which want to use a specific widget class.

When a new class is implemented, it defines its own class data structure, containing its additional attributes and methods prototypes. To *inherit* the data of its base class, it simply includes the data structure of its base class as its *first* attribute. After that the new class can define any new attributes that it wants to use. This mechanism ensures a memory layout like shown in figure 5. The first data attributes in a widget object are always the attributes of the RMP generic widget class and the RMP core can handle widgets of any type.

To access attributes, each toolkit provides per-widget-class macro definitions that cast a given widget object to the appropriate type. Widget methods can use these macros to access attributes of the widget classes themselves or of the widget's base classes. The macros are used with the syntax:

```
TOOLKITNAME_WIDGETTYPE_DATA(widget)->attribute
```

where

widget widget handle

attribute attribute in the widget class

As the access macros assume a known offset of the attribute data within the widget object, it is *not* possible to inherit data from multiple base widget classes.

2.5.2 Private data

When designing an object-oriented class hierarchy, private data is a way to protect attributes of a base class from accesses by a derived class. In object-oriented languages like C++, when a base class defines private attributes, a derived class inherits both, private and public attributes of the base class, but can not *see* or *manipulate* the private ones. A C++ compiler supports this mechanism.

While implemented in the C language, the inheritance mechanism used by RMP *does not* provide C++-like implementation of private data in classes. However it is possible to implement the concept of private data in widget classes. When the instance of a class is created, the implementation can allocate memory to store its private data.

Instead of defining the private data structure in an header file, it is defined only in the C-file where it will be referenced. This means, that these attributes *can not* be inherited by other classes.

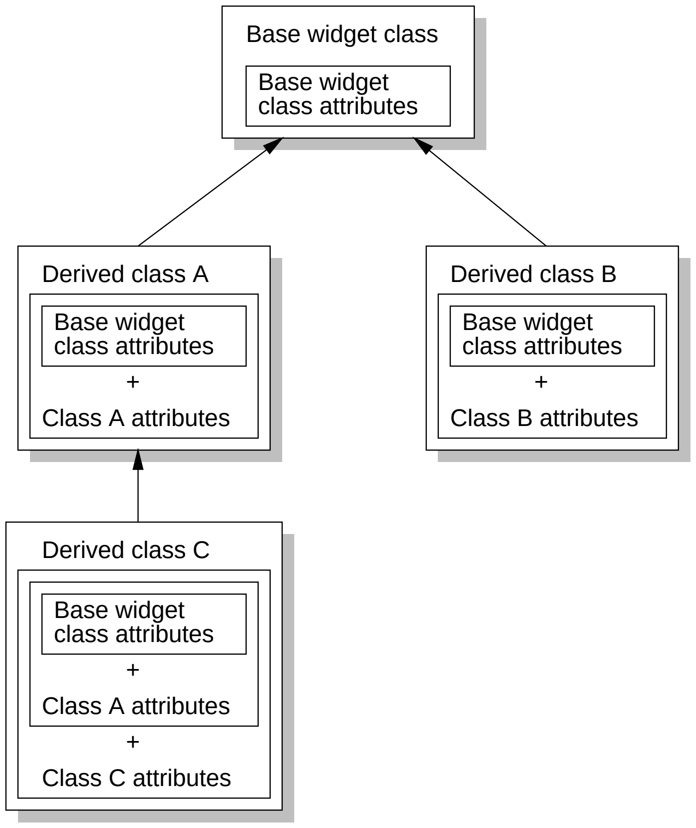


Figure 5: Data inheritance in widgets

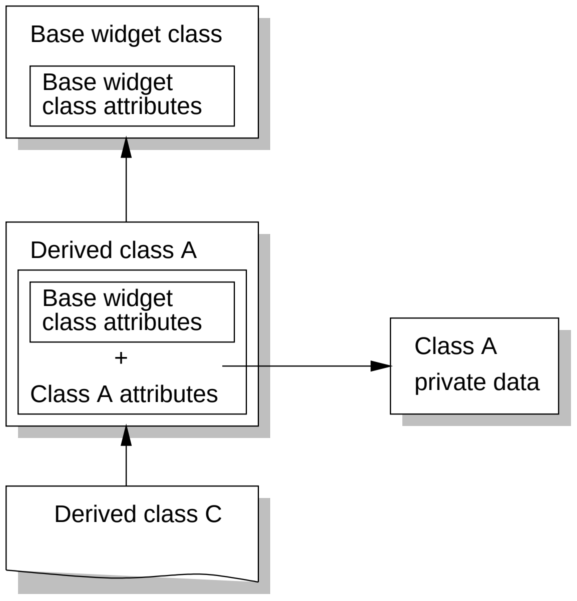


Figure 6: Private data in widgets

2.5.3 Virtual methods

As described earlier in this section the RMP generic widget class implements a common or generic behaviour. A part of this is implemented in form of *virtual methods*, which are registered in the widget object when the widget is created by the user application. When a widget class is derived from the RMP generic widget class it inherits both data and generic behaviour. If the derived widget does not change these entries, it automatically inherits these methods, as the RMP core will use the virtual method entries in the widget object to call the widget's virtual methods.

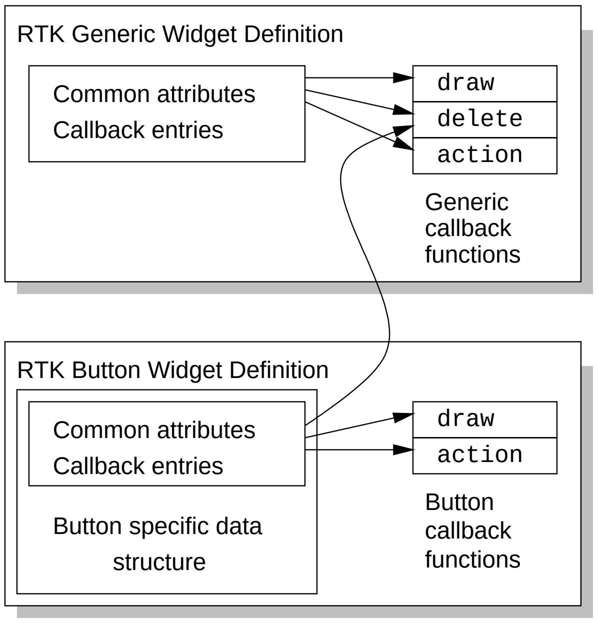

RTK Generic Widget Definition
Common attributes
Callback entries
draw
delete
action
Generic callback functions
RTK Button Widget Definition
Common attributes
Callback entries
Button specific data structure
draw
action
Button callback functions

Figure 7: Example for widget method inheritance in RTK

New widget classes can extend their functionality by implementing their own virtual methods and registering the in the widget object, thus overwriting the entries of the base class. Figure 7 shows this mechanism for RTK. A new class can also introduce new virtual methods which will be inherited by its derived classes.

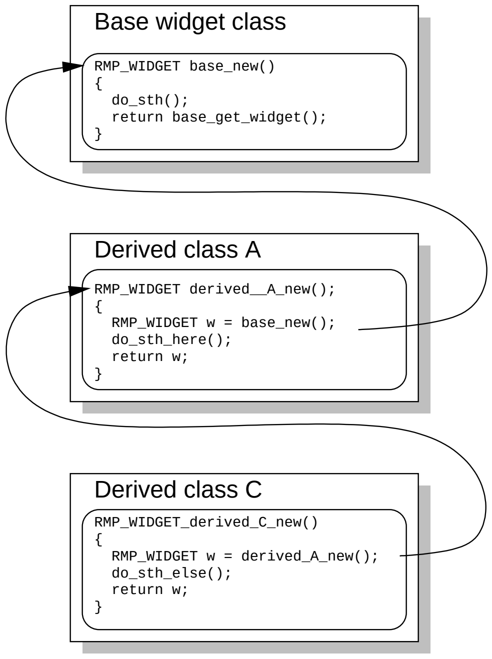
When a new widget class overwrites a virtual method, it might be necessary to remember the original virtual method. For example, when overwriting the *delete* method, the widget *must* call the *delete* function of its base class within its own *delete* method. The *delete* method is the only method where this is required.

Virtual methods of widget classes are *not visible* to the user application as they are only registered as callback entries in the widget objects. The user application *is not* able to access any virtual method of a widget class directly. How user applications can use these methods is described in section 3.4, page 30.

2.5.4 External generic class methods

Not all generic functionality of a widget class is implemented in form of virtual methods. There also exist external API methods , for example, the *new* method of a class is such an external API method. These methods are defined in the header file and are only used by *derived class implementations*. If necessary, the widget class may provide a macro definition for accessing the *new* method in its header file.

External generic class methods are not implicitly inherited like virtual methods, nor can they be overwritten, they need to be implemented by the derived class. The derived class knows about its base generic methods class and can call the appropriate external generic class method. Figure 8 shows this mechanism. It requires disciplined coding when implementing new widget classes.



```
RMP_WIDGET base_new()
{
  do_sth();
  return base_get_widget();
}
```

```
RMP_WIDGET derived__A_new();
{
  RMP_WIDGET w = base_new();
  do_sth_here();
  return w;
}
```

```
RMP_WIDGET_derived_C_new()
{
  RMP_WIDGET w = derived_A_new();
  do_sth_else();
  return w;
}
```

Figure 8: Inheritance of internal class methods

2.6 RMP widget generic API

The user application can apply certain generic functions on any widget within an RMP toolkit. To provide common attributes for all widgets is one step to achieve this goal. Another step is to implement a common behaviour for all widgets using a generic API. In this way, not only does the application need no knowledge of the internal attributes of a widget, it can also count on a pre-defined behaviour or reaction to external stimuli, such as events.

The generic behaviour is implemented using *virtual methods* and *external class methods*. All widgets must implement these methods or inherit them. These methods have a defined return type and parameter list which is used throughout all RMP toolkit widgets. The generic behaviour of the generic RMP widget is very limited. It is basically used and extended by widget classes of other toolkits while specializing them for the toolkit's needs. The RMP generic widget defines *three* generic methods:

- Create a widget (external class method)
- Delete a widget (virtual method)
- Handle an action (virtual method)

The generic behaviour is implemented in the file:

```
rmp_widget.c
```

The reason for having two different ways of implementing generic behaviour is that some methods like *delete* and *handle action* are independent of the widget type. The RMP core does not need to know about the widget type when calling these methods. However, when calling a *new* method, the type of the widget is important and must be provided.

2.6.1 Create a widget

When a new widget is created, several things have to be done:

- Allocate memory for the widget object
  When an application wants to create a widget, memory has to be allocated to hold the attributes of the widget's class definition. This *always* happens in the RMP generic widget class *new* method.
- Register virtual methods in the widget object
  The default virtual methods are registered in the widget object by the RMP and may be overridden as the widget instance evolves.
- Set the type of the widget
  The default type (`RMP_TYPE_WIDGET`) is set for the new widget.
- Initialize data fields and buffers to default values
  Default values for *parent*, *child* and *next* widget are set.

The generic widget *new* method takes care of all these thing and prepares a new widget object to be used. Individual widget *new* methods use the generic widget *new* method to request a new widget object and then refine the parameter by overwriting the appropriate values. The common syntax for widget class *new* methods is:

```
RMP_WIDGET toolkitname_widgettype_new(uint wsize,
                                      optional params,
                                      const char *name);
```

where

`wsize` the size of the widget object (see section 2.7.1)

`optional params` list of parameters for the *new* method

`name` the name of the widget

The number and type of *`optional params`* for the widget depends on the type of the widget.

However, the *new* method of a widget class *must not* be used by the user application. To create a widget object every widget class provides a macro definition that is used by the user application. The macro is defined in the header file of the widget class:

```
TOOLKITNAME_WIDGETTYPE_NEW(optional params, const char *name);
```

The `wsize` parameter is omitted in the macro definition. The appropriate size of the individual widget object will be filled in by the macro definition. The user application does not have to know about the actual widget's size.

2.6.2 Delete a widget

The generic widget *delete* method takes care of freeing the allocated widget object memory. In individual widget implementations this method can be used to take actions, when the widget will be deleted, such as freeing widget dependent, locally allocated memory or do something else. It is even possible to send out an action to another widget.

The *delete* method uses the following syntax:

```
static void toolkitname_widgettype_delete(RMP_WIDGET w);
```

where only the parameter list is mandatory. The naming can differ, as this method actually is defined static in the C module. However, it makes sense to keep a consistent naming scheme.

When a new widget class implements its own *delete* method and overwrites the virtual method entry of its base class in the widget object, it has to take care that the *delete* method of the base class is called at the end of its own *delete* method. Therefore the widget has to keep a reference of the base class *delete* function.

2.6.3 Handle an action

This method is also defined empty for the generic widget, except it returns a

```
RMP_NOT_HANDLED
```

code. Other widgets use this method to implement their action handlers.

Action handlers look like:

```
static void toolkitname_widgettype_action(RMP_WIDGET w,
                                          uint action,
                                          void *data);
```

where

w the widget to receive the action

action the action code

data optional data structure

Individual widget action handlers can indicate a success by returning:

```
RMP_HANDLED
```

Again, as with the *delete* method, only the parameter list, not the naming is mandatory.

2.7 Memory management

When widget objects are created or deleted, memory has to be allocated or freed. RMP defines a mechanism to ensure a safe memory management for RMP widget classes.

2.7.1 Allocating memory

When the user application wants to create a widget it calls the *new* method of the specified widget class. The mechanism described in section 2.5.2 causes the RMP generic widget *new* method to be called. This method needs to know about the size of the actual widget object. Therefore the *size* of the widget object must be passed through all *new* methods up to the RMP generic widget *new* method. This is necessary because the individual widget *new* method only knows about the object size.

2.7.2 Freeing memory

When a widget object is deleted by the application, the associated memory buffers have to be freed. This can be private data structure buffers which have been allocated by individual widget implementations or dynamically, during run-time allocated data. Additionally the memory for the widget object itself has to be freed.

The RMP core always calls the *delete* method of a widget when it should be deleted. The individual widget *delete* methods take care of freeing their own allocated data before calling the *delete* method of their base class. This way allocated memory is successively freed during the delete procedure.

Finally, the *delete* method of the RMP generic widget class will be called, freeing the memory of the widget object.

2.8 Widget communication via actions

RMP extends the ROME communication model through a major innovation in its design, the concept of widget–widget communication. Widgets communicate using *actions* which are completely separated from ROME *events*. These actions are directed to specific widgets invoking the widget's *action handler*. As widget–widget communication is *only* based on actions, they implement their own communication mechanism which is completely independent from the rest of the system.

Introducing this new layer of communication opens a wide variety of possibilities for widget communication and interaction. Allowing widgets to send actions to other widgets makes it possible to build *chains of activity* within widget structures that can complete complex tasks. As widgets do not know where actions actually come from, they can implement a context independent behaviour — they will always behave the same way, no matter who sent a specific action. Figure 9 shows an example scenario where a widget receives the same action from three different sources.

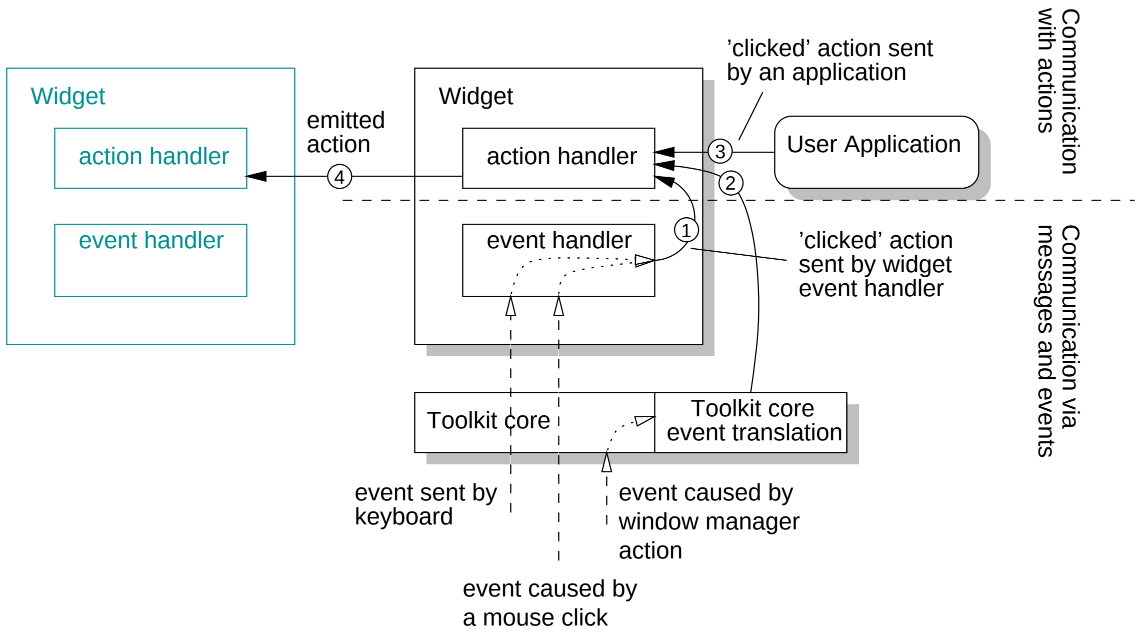

Widget
action handler
event handler
emitted action
4
Widget
action handler
event handler
'clicked' action sent by an application
3
User Application
2
1
'clicked' action sent by widget event handler
Communication with actions
Communication via messages and events
Toolkit core
Toolkit core event translation
event sent by keyboard
event caused by window manager action
event caused by a mouse click

Figure 9: Widget event-action mechanism

The mechanism of event translation is described in section 3.9, page 35.

2.8.1 Sending an action

Actions can have different origins. They can be sent by:

1. The widget itself (see section 4.2.8)
2. Lower layer processes or libraries
3. User applications
4. Other widgets

To send an action the RMP generic API provides the function:

```
void rmp_action(RMP_WIDGET widget, uint action, void *data);
```

This function *must* be used when sending actions to a widget.

2.8.2 Action structure

An action usually consists of a single command code and an accompanying data structure. The command code is a integer value which is defined in the toolkit's external header file (see section 3.7.3, page 34). Action command codes (or actions for short) are named depending on the toolkit and widget names:

```
#define TOOLKITNAME_AC_WIDGETTYPE_CLICKED  0x100
```

When sending an action a data structure can also be passed to the widget. Whether or not a data structure must be defined depends on the widget type and the widget implementation.

If a data structure is defined it is *always passed by reference*, even it is a simple data type like `int` or `char`.

For example:

```
int x = 23;
rmp_action(widget, TOOLKIT_AC__WIDGET_DUMMY, (void *) &x);
```

has to be used instead of:

```
rmp_action(widget, TOOLKIT_AC_WIDGET_DUMMY, (void *) 23);
```

as the size of the expected data type and a pointer data type *can not* be assumed to be equal.

2.8.3 Action purpose

An action is the cause for the widget *to do something*. Usually it changes its state or appearance, or both. Furthermore the *action* is forwarded to an user callback function, if one has been defined by the user application. This way it is possible to *connect* different widgets within a user interface to automate the user interface behaviour.

2.8.4 Using actions for inter-process communication

As it is possible to resolve a widget handle using its name, an application can get a handle for any widget in the system. It is possible to get handles of widgets that are used in other processes or applications. This way it is possible to send actions between widgets of different processes and pass information between them. This mechanism provides an easy way of inter-process communication using widgets.

There is, however, a drawback. As actions are sent using API calls, the widget which receives the action runs in the process context of the process which contains the sending widget. Thus, care must be taken when implementing such action callbacks in user applications.

2.8.5 User callback function support

User callback functions can be connected to individual widgets. This way the user application will be informed of any action that occurred in a widget. The callback function will be provided with useful information such as:

- The handle of widget that caused the callback
- The type of the action
- A user defined opaque parameter

Each widget only allows the definition of *one* user callback function. So the application has to take care to demultiplex the different actions that can occur in a connected widget.

3 Toolkit layer

User applications use widgets of different toolkits to implement functionality like building a graphical user interface or managing sound samples. To do so, they need an API that provides all functions to create, manage and delete widgets of different toolkits. The toolkit layer integrates all available toolkits in the system, building a working environment for the user applications. To the user layer, it presents a merged API of all toolkits available in the system.

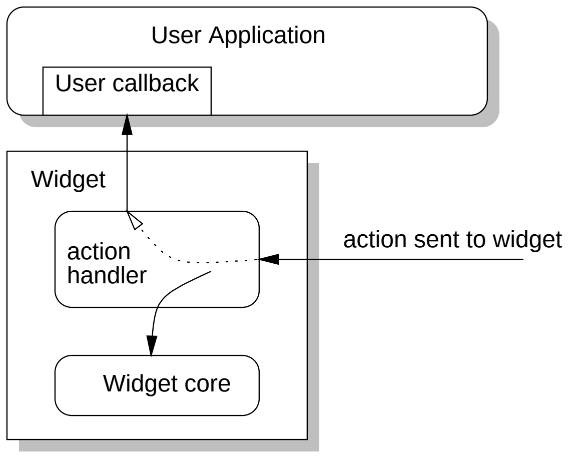


Figure 10: Widget user callback

3.1 Toolkits

Toolkits are basically a collection of widget classes around a toolkit core, providing a easy-to-use user application interface. Toolkits in RMP are designed to work seamlessly in the RMP framework, following the implementation rules of RMP.

Toolkits use widgets to implement their main functionality while the toolkit core implements the *management* of the toolkit's widgets.

3.1.1 Modularity

One advantage of having toolkits instead of integrated functionality in a monolithic RMP is, that it allows toolkits to be modular. Toolkits for different purposes can be integrated into RMP independently providing a flexible mechanism for building a hardware adapter application. On the other hand, the *action* communication mechanism (see section 2.8, page 25) allows the connection of widgets of different types in different toolkits without the need of hardcoded message pathways, making the system even more flexible.

3.1.2 Extendibility

Another advantage of using toolkits is that they consist of a hierarchy of widgets, which themselves are mostly independent of each other. By adding new widget classes to an existing toolkit it is easily possible to extend the functionality of an toolkit without changing other code in the toolkit. By using *pre-defined abstract widget classes* to derive new widget classes, the development of new functionality is made extremely easy.

3.1.3 High level system abstraction

Toolkits introduce another layer of abstraction to the system. Using widgets, they can provide a high level abstraction of system components, encapsulated in an easy-to-use framework. The abstraction of the system into widget objects allows a hardware independent application development which results in *very* portable source code.

The object-oriented design of toolkits makes it possible to create object oriented programs while using the C programming language. The application can take advantage of both the efficiency of the C language and the object abstraction provided by the toolkit.

3.2 RMP generic framework

To define a generic framework, RMP provides the *RMP generic toolkit core API* and the definition of the *RMP generic widget class*. The RMP generic toolkit core provides the basic functionality of managing widgets. Other toolkit cores will interact with it to integrate themselves into RMP. The *RMP generic widget class* (described in section 2.3, page 17) defines the class anchor for all widget classes in toolkits.

Using the RMP generic framework, developing new toolkits is very easy. All common function are already provided by RMP and can be used by new toolkits. Thus, toolkits only need to extend the RMP functionality, not re-implement everything.

All C-modules of the RMP generic framework are located in a ROME-module, called RMP. This ROME-module is a prerequisite for *any* toolkit module used with RMP.

3.2.1 RMP core

The RMP core is the central access point within the toolkit layer. It contains a list of all *root widgets*[2] in the system and implements all the functionality needed to manage and maintain this list. Toolkit cores and toolkit widgets as well as user applications interact with the RMP core in order to address other widgets in the system. When widgets are created by applications they can be combined in more complex structures. The *root widgets* of these structures will registered in the RMP core *root widget* list which gives the RMP core complete control over all widgets in the system.

The RMP core is implemented in a single C-module, called:

```
rmp.c
```

The RMP core only provides a part of the functionality of the whole RMP generic framework. The RMP generic widget class provides the base class for all widgets in RMP. It is described in section 2.3, page 17.

3.2.2 Header files

RMP provides two header files. One is for use by user applications and defines the prototypes of the API functions as well as the data definitions and external data types. This file is called:

```
rmp.h
```

The other file is only used by toolkits and defines internal data structures, types and the prototypes of the internal RMP core functions. The file is called:

```
rmp_internal.h
```

User applications should not include the internal header file when they are using RMP functions. This prevents user applications from becoming dependent of the internal architecture of RMP. By including only the external header files, the user application stays independent of the implementation details of RMP.

3.2.3 Data widget

Additionally to the RMP generic widget class, which is an abstract class, RMP also provides the *data widget class* from which widget objects can be created. The purpose of the data widget is to provide a generic data container objects which can be used by user applications to share or send information. An application (or process) can use it to store data and another application can read or change them.

As the data widget class is derived from the RMP generic widget class it supports the *action* and *user callback* mechanism. User applications use *actions* to manipulate data within the data widgets and the data widget itself uses the *user callback* mechanism to forward the *action*. This enables user applications to *watch* the contents of certain data widgets.

The data widget class does not restrict the type of data being stored in the widget objects. Therefore the widgets are very versatile and can be used, for example, as common data pools for several processes or an information relay within a widget object tree.

[2]The term *root widget* refers to the widget in a widget tree which has no parent.

3.3 Toolkit integration

RMP provides a generic framework which makes the integration of new toolkits very easy. To achieve an efficient implementation, RMP toolkits are structured a specific way to make sure they fit into the environment. They consist of several components that implement different, well defined, functionality. These components can be divided into three main groups:

- Toolkit core, implementing both the toolkit core API and the widget virtual methods APIs
- Generic widget class implementation
- Sets of individual widget class implementations

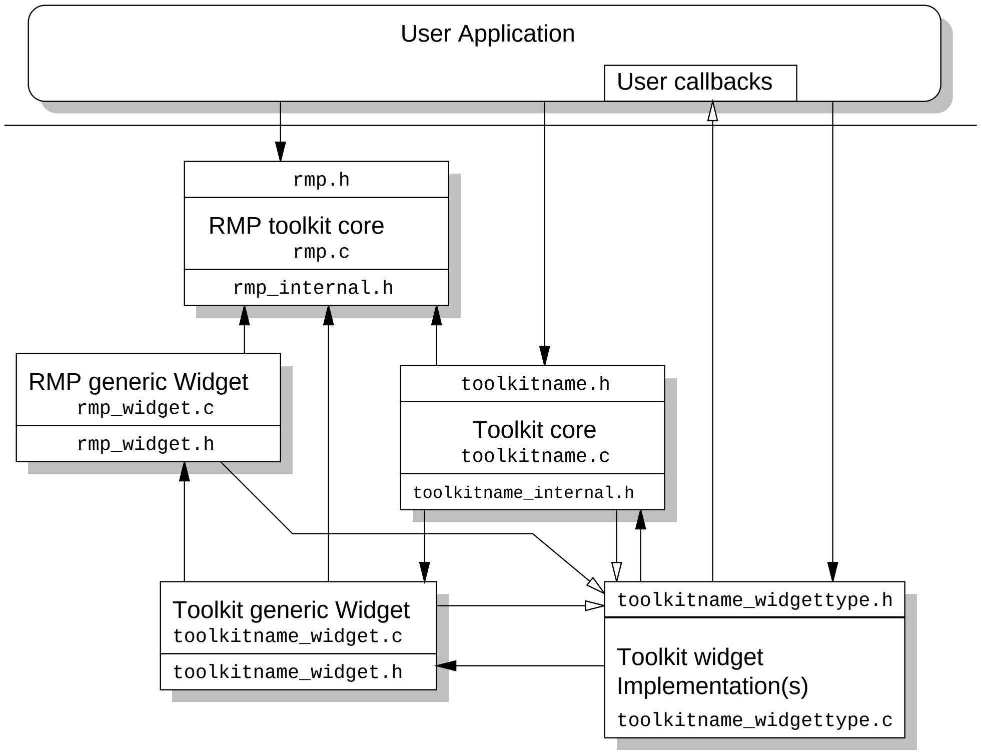


Figure 11: Toolkit implementation overview

Figure 11 shows an overview of a individual toolkit integrated into the RMP toolkit framework with the RMP toolkit core and the RMP generic widget class.

3.3.1 Basic components

The components of a toolkit are usually implemented in C-modules[3]. These C-modules follow a naming scheme that is consistent throughout all toolkits for ROME:

***toolkitname*.c:** Toolkit core;

***toolkitname*.h:** Toolkit external header file;

***toolkitname*_widget.c:** Implementation of the *generic widget class* of the toolkit;

[3]C-modules are *not* ROME modules. A C module is a single C-source file whereas a ROME module can contain several C-modules to build a process or a shared library

***toolkitname*_widget.h**: Internal header file of the *generic widget class* of the toolkit;

***toolkitname_widgettype*.c**: Individual widget implementations;

***toolkitname_widgettype*.h**: Individual widget header file used by user applications and widgets which are derived from that individual widget.

Each C-module implements a specific functionality. Within the toolkit the C-modules are kept as independent as possible and communicate through well defined interfaces. The modular design guarantees that a toolkit will be easily extendable.

3.3.2 Message and event handling

All toolkits implement an event handler for handling events from the *adapter layer*. As toolkits are shared libraries, they cannot implement a queue handler and are not able to receive messages or events directly. Instead, the process that uses the toolkit will receive those messages and events. To allow the toolkit to handle them, the application has to call the handler functions of each of the toolkits it is using and pass down *every* reply message or event to be checked by the toolkit handler.

Each toolkit handler returns a status, indicating whether it handled the message (`RMP_HANDLED`) or event or not (`RMP_NOT_HANDLED`). If so, the user application *must not* process it any further. Otherwise it *must* pass it to the next toolkit handler and so forth. If the message or event was not processed by *any* of the toolkit handlers, the user application should try to handle it itself.

3.4 Toolkit core

As described in section 2.3, RMP provides a *generic widget class* to define a generic anchor for all toolkit class definitions in RMP. As all toolkit widget classes are directly or indirectly derived from this class, they all inherit the same generic data and behaviour.

Every toolkit implements its own core, defining the generic API and an API for the virtual methods of its widgets. The toolkit core is implemented in the:

toolkitname.c

C-module. Every new toolkit core extends the existing toolkit API of RMP.

3.4.1 Parameter shielding

User applications can not access the toolkit widget class *virtual method*s directly, they always have to use toolkit core functions. The toolkit core provides both its own *generic* API and an API for all *widget class virtual methods*. It redirects virtual method calls from user applications to the individual widget. This way it is possible to check for invalid parameters in the toolkit core rather than in the widget class itself, which gives the advantage of having a *trusted* run-time environment at the widget layer. This avoids the necessity to run checks in *every* widget class, leading to smaller code size and less run-time overhead.

3.4.2 Virtual method multiplexing

Another important task of the toolkit core is to multiplex the calls of virtual methods in the widgets. Within a toolkit widget class hierarchy it is possible that new widget classes add new virtual methods. These methods are only valid for the new widget class and all its derived classes, not for classes in higher levels of the class tree. So it is possible that some widget classes provide a certain virtual method entry, others do not.

The toolkit core provides an API for *every virtual method* that is implemented by its widget classes. As the user application uses the toolkit core to access widget's virtual methods the toolkit core needs to know if the request is valid for the widget type, deciding if the call should be executed or not. The toolkit core uses the *type* field in the widget object to determine its type and to check if the specified widget implements the requested virtual method.

This mechanism makes it necessary to change the toolkit core code when adding a new widget class that introduces new *virtual methods*. If the new widget class does not introduce any new *virtual methods*, no changes in the toolkit core are necessary.

3.5 Toolkit widget implementation

Every toolkit defines a set of widget classes. The toolkit always derives its own *generic widget class* from the *RMP generic widget class* to inherit all the generic data and behaviour and to extend it by adding its own attributes and methods.

3.5.1 Generic widget class

The *toolkit generic widget class* is the anchor for *all* widgets in the toolkit. It defines the *generic* attributes and behaviour of *all* widgets of the *specific* toolkit. Every widget class in the toolkit is directly or indirectly derived from this class. The toolkit generic widget class is implemented in the file:

```
toolkitname_widget.h
```

All generic API method names of this class also use this prefix.

3.5.2 Other widget classes

Additionally a toolkit implements a set of widget classes. Usually, every widget class definition is put in a separate C-file called:

```
toolkitname_widgettype.c
```

All functionality of the widget class is implemented in this single file, including the *new* method, the *virtual methods* and the *generic API* as well as specific external support methods for the individual widget type. All external API methods of a widget class use the filename as prefix.

3.6 RMP core generic API

The RMP core API provides both the generic API for managing widgets as well as the API to *shield* widget class calls. The API contains functions to:

- Nest widgets
- Lookup widget names (finding handles of widgets)
- Set the user callback function of widgets
- Delete widgets[4]
- Send actions to widgets

3.6.1 Nest widgets

When widgets are created, they are *not* integrated into any structure by default, they are created as *single* objects. To build more complex widget structures which provide more functionality it is possible to combine widgets of different types. The basic function to do that is the widget RMP *add* function:

```
void rmp_add_widget(RMP_WIDGET parent, RMP_WIDGET child);
```

Given a *parent* and a *child* widget, it adds the child widget into the child list of the parent widget. The *add* function can be applied to *any* pair of widgets. This method of combining widgets is called *nesting*.

As only the RMP core provides an API to nest widgets and all widget references are kept inside the RMP core, it is possible to find *any* widget of *any* type that currently instantiated in RMP. This, of course, requires that the user application registers its widgets in the RMP core by adding the *root* widget of its widget tree.

[4]Creating widgets is done by using the individual widget's API.

3.6.2 Lookup widget names

When an application wants to apply a function to a widget it needs to know its widget *handle*. When an application creates a widget, it can keep the handle for its later use. however, it is not always possible to use stored handles. For example, when creating a complex widget (see section 4.2.4) the application can not see the widget handle of the nested widgets inside complex widgets. Or the application may need to send an action to a widget in another process.

In these cases the application does not know about the widget handle, but it knows about the widget name. Complex widgets define a fixed naming scheme for their nested widgets. Once the application knows about the complex widget's name, it can build the name and lookup the widget handle of the nested widget.

The function to do so is called:

```
RMP_WIDGET rmp_lookup_widget_name(const char *widget_name);
```

and returns the handle to the widget if it was found or `NULL` if the name does not exist or is invalid.

3.6.3 Set user callback function

In some cases user applications need to know when their widgets receive *actions* in order to react to certain events in the system. It is possible to set a user callback function to get informed of any actions that occur in a widget. To set the callback function the use application uses the:

```
void rmp_set_callback(
                  RMP_WIDGET widget,
                  void (*user_cb) (RMP_WIDGET, uint, void *),
                  void *data);
```

function.

The user application has to provide a callback function which has the prototype:

```
void user_cb(RMP_WIDGET widget, uint action, void *data);
```

Currently it is only possible to set *one* callback function per widget. The widget will send *all* actions to that callback function, so the user callback function has to de-multiplex the actions by itself.

3.6.4 Delete a widget

When widgets are no longer needed they can be deleted from the widget list. Deleting a widget always happens through the:

```
void rmp_delete_widget(RMP_WIDGET widget);
```

function, which calls the widget *delete* virtual method and removes the widget from the widget list.

To avoid *orphans* in the widget tree, the RMP core delete function takes care that all child widgets of the deleted widget are deleted as well. This means, that the *whole widget tree* under the deleted widget will be removed from the widget list and all *delete* methods of these child widgets will be called.

3.6.5 Send actions to widget

As described in section 2.8, widgets can communicate via actions. It is also possible for user applications to send an action to a widget. For sending actions to a widget the:

```
void rmp_action(RMP_WIDGET widget, uint action, void *data);
```

function is provided. The value of `action` is defined in the individual toolkit's *external* header file and depends on the widget type, and on what the widget should do. The `data` parameter may need to point to a data structure which can be used by the widget and is therefore also dependent on the individual widget implementation.

3.7 Toolkit core generic API

A toolkit *must implement* a set of mandatory generic interface functions. These functions follow a naming scheme and have a defined behaviour and functionality so they are easy to identify within *any* toolkit in RMP. They are implemented in each *individual toolkit core*:

- Initialize the toolkit
- Message or event handler

Of course, toolkits can implement many other generic functions (as e.g. RTK does) for their widgets, but they *must* at least provide the ones shown above.

One might argue that a programmer could choose whatever implementation style is suitable for a new toolkit and the toolkit widgets as the only thing the toolkit has to provide in the toolkit is the generic API. This is probably true, but following a common RMP implementation scheme for all parts of the toolkit makes it much more consistent and easier to use.

3.7.1 Initialize the toolkit

A toolkit keeps a lot of data that has to be maintained during run-time. Usually, before a toolkit can be used its internal data structures or state must be initialized. For this task the:

```
int toolkitname_init(void);
```

function is provided. Each process using the toolkit *must* call this function *before* usage of *any other* toolkit function. it ensures that the toolkit is put in a defined state. The functions returns 0 for success, -1 if an error occurred.

3.7.2 Handling messages and events

As toolkit modules are not processes they have no message queues and can not *receive* messages directly. However they might send messages to the *adapter layer* which will be returned after being processed, or call API functions in the *adapter layer* that trigger events to be sent back. For this reason every toolkit implements an event handler which must be called by the user application once it receives a message.

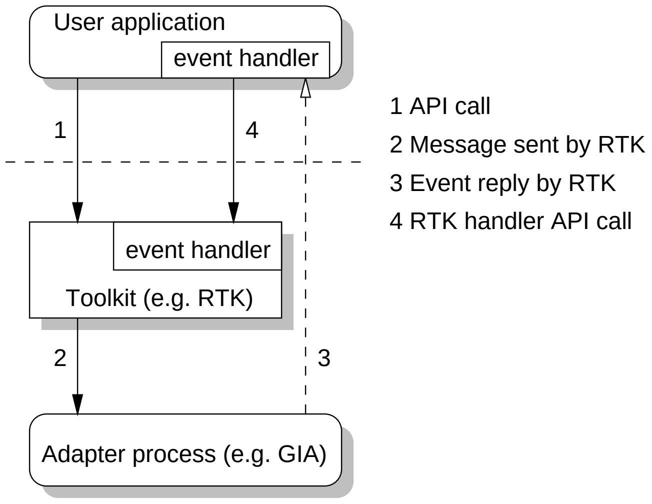


Figure 12: Message/event handling in the toolkit

The toolkit event handler will determine the message type, and handle it, if necessary. It returns a status to the application to inform it whether the message has been handled or not. Figure 12 shows this mechanism for RTK and GIA. The event or message handler has the format:

```
int toolkitname_handler(ROME_MESSAGE *msg);
```

and returns

```
RMP_HANDLED
```

or

```
RMP_NOT_HANDLED
```

to indicate the status of the operation.

3.7.3 Toolkit header files

A toolkit core must provide two header files. One of which is used by the user applications and contains all API definitions, pre-processor defines and type definitions that are externally used by the toolkit. The file is called:

```
toolkitname.h
```

The other file contains definitions that are user *internally* by widget class implementations of the toolkit. It contains the internal API for use by widget classes and other internal definitions. The file is called:

```
toolkitname_internal.h
```

3.8 Toolkit widgets API

One of RMP's goals is to keep a consistent naming scheme for its API. This requires that all toolkit implementation follow certain rules for naming the *generic* API functions. This is also a requirement for the generic API methods of widget classes.

However, RMP currently defines only a single *generic* method for widgets, the *new* method, which has to be implemented by all RMP toolkit widget classes.

3.8.1 Create a widget

Toolkit cores do not provide a generic method to create widgets, because the widget *new* methods are depending on the widget type. Individual widget implementations might require different parameter types or different numbers of parameters depending on the complexity of the widget. Although it is possible to create a *generic* method to suit the needs, lack of parameter type checking generally makes this approach more complex.

To keep a simple API the widget *new* methods is provided by every individual widget implementation (see section 2.6.1, page 22). This implies that the user application has to include the header file of each widget type it uses.

For abstract classes (see section 2.2.2, page 16) it is not necessary to define an *external* reference to the *new* method in form of a macro. For classes of which an application can create instances, such a macro definition is mandatory.

3.8.2 Individual widget API

New toolkits can introduce a set of new functions with their widget classes. In the case of virtual methods or generic API functions they have to follow the rules of inheritance defined by RMP in order to work properly within the RMP framework.

New *generic* functions, other than the *new* function are defined as external accessible functions and can be directly accessed by the user application. This makes it necessary that these functions check for invalid parameters as there is no protection provided by the toolkit core[5].

New *virtual methods* must follow the RMP implementation rules (see section 2.5.3, page 21). This is the only way to ensure that all toolkit widgets will work properly.

Additionally a new widget class can implement a set of methods that supports specific functions of that widget type. These methods also have to be accessible for the user application and have to check their parameters at run-time.

[5]Of course, a new toolkit *could* integrate *all* its widget functions into the toolkit core and provide protection for its widgets.

3.8.3 Widget header files

Every widget class implementation provides a header file. The header file contains all type definitions, pre-processor defines and macros that are associated with the widget class and are for use by user applications. It also contains the *new* macro discussed earlier. The header file is called:

```
toolkitname_widgettype.h
```

This header file also contains information that is provided for other widget classes that derive from this class. It contains, for example, the data structure of the widget class and type definitions.

3.9 Event to action translation

As described in section 2.8, widgets communicate via actions. In ROME, however, processes usually communicate via *messages* or *events*, which are not directly understood by widgets.

To be able to react to *events* received from the *adapter layer*, these events are translated into actions at *toolkit* level. Where this translation from events to actions exactly happens is not important. The only thing of importance is the resulting action that is sent to the widget.

However, following the object-oriented design of RMP, the translation mechanism is usually implemented in the widget code itself. At first sight this might seem strange, but the reason to do so is the way events are translated. Events are not always mapped to actions one by one. To create, for example, an *clicked* action for a widget, a series of events is necessary. The event translation algorithm has to keep track of the events and finally create the appropriate action, if a certain event pattern occurred. For the *clicked* action, the following events have to occur:

1. Button push event inside the widget area
2. Any number of other events, except button release event
3. Button release event inside the widget area

Only after all the conditions are satisfied, the *clicked* action will be created. As the requirements for action can vary with each individual widget type, each widget implements its own event translation algorithm. This increases the flexibility of creating new widgets.

4 RTK - Rendering ToolKit

Graphical user interfaces (GUIs) are a way to make the interaction between user and application easier. They provide an *obvious* way of using application features by using graphical elements that can also be found in the real world. To achieve an consistent look and feel, graphical user interfaces should use the same graphical elements throughout different applications.

The Rendering ToolKit is designed to provide features for doing this in RMP. It implements widgets to build graphical user interfaces, providing an abstraction for various control elements of different types. These elements currently are:

- Windows
- Plain buttons
- Buttons containing labels
- Buttons containing pixmaps
- Buttons containing labels and pixmaps
- Togglebuttons
- Pixmapped buttons

- Pixmapped togglebuttons
- Sliders
- Pixmaps
- Canvases
- Drawareas
- Labels
- Text input fields

These widgets can be used to build a GUI. They are completely integrated into the RMP widget framework, building a subset of the RMP generic widget class. All RMP functions can be applied to any RTK widget.

4.1 RTK overview

RTK is located in the *toolkit layer*. RTK manages all windows and widgets created by user applications. RTK interacts with the *adapter layer* (GIA) as well as the user application. An overview of RTK in an application context is shown in figure 13.

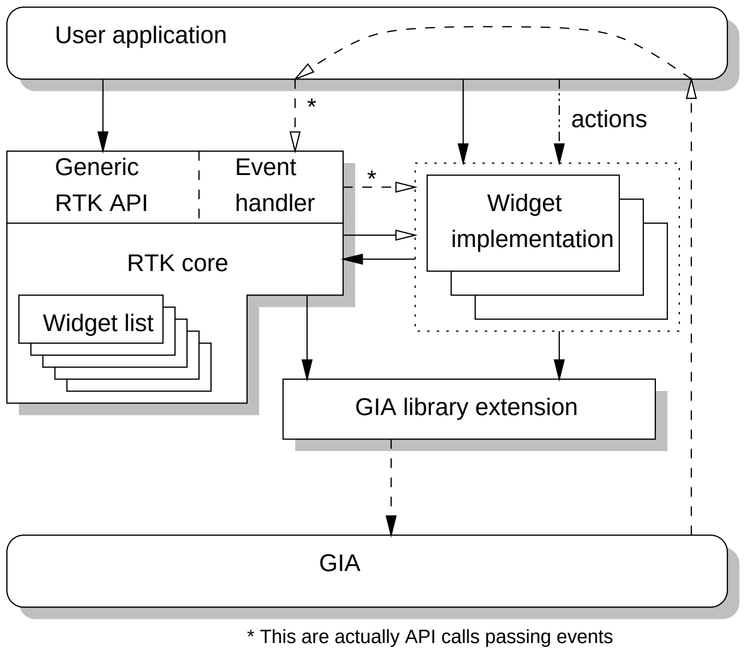


Figure 13: RTK overview

RTK is a *shared library* and is located in the *toolkit layer.* As it interacts with the *adapter layer* (GIA in particular), events are sent back that have to be processed by RTK. The user application passes any event received to the RTK event handler (see section 3.7.2). The RTK handler analyzes the event and if appropriate forwards it to the widget that currently has the focus.

If the event was not handled by RTK, the user application can handle the event in its own handler afterwards. The RTK handler operation is completely independent of any user application event processing.

4.1.1 RTK and GIA drawables

RTK uses GIA drawables to draw widgets on the screen. New drawables are only allocated for *window* widgets, other widgets like buttons or labels are drawn into the drawable of their window widget. This technique has several advantages:

- Only one drawable is needed for each window which reduces memory usage;
- Fewer drawables result in less overhead and higher performance;
- By keeping the whole window content in a single drawable the GIA can take care of clipping and redrawing exposed windows.

RTK only keeps *references* of GIA drawables within its widgets, the *actual* drawable buffer is kept in GIA. When RTK deletes a window object containing a drawable reference it also requests GIA to delete the drawable.

4.1.2 RTK color configuration

RTK can be configured using the system configuration library (see section 14.3, page 73). It is possible to define the following attributes of RTK widgets through entries in the system configuration database:

- Button color (`COLOR_BUTTON`)
- Button shadow color (`COLOR_BUTTON_DARK`)
- Button light color (`COLOR_BUTTON_LIGHT`)
- Slider color (`COLOR_SLIDER`)
- Slider shadow color (`COLOR_SLIDER_DARK`)
- Slider light color (`COLOR_SLIDER_LIGHT`)
- Label text color (`COLOR_LABEL_TEXT`)
- Label background color (`COLOR_LABEL_BACKGROUND`)
- Text input field background color (`COLOR_TEXT_INPUT_BACK`)
- Text input field focused frame color (`COLOR_TEXT_INPUT_FRAME_FOCUS`)
- Text input field unfocused frame color (`COLOR_TEXT_INPUT_FRAME_NOFOCUS`)
- Text input field text color (`COLOR_TEXT_INPUT_TEXT`)
- Text input field focused cursor color (`COLOR_TEXT_INPUT_CURSOR_FOCUS`)
- Text input field unfocused cursor (`COLOR_TEXT_INPUT_CURSOR_NOFOCUS`)

The configuration database is used by *all* widgets. If an entry in the database is changed, all application widgets will be affected. However, they will not be automatically updated on changes to database entries.

4.2 RTK widget classes

RTK defines a set of widget classes for creating RTK widgets. Applications create widgets through RTK widget API calls and usually RTK widgets are graphical elements that appear on the screen. A RTK widget is a very versatile object. It can be very simple and just contain some data or implement more complex graphic elements such as a labeled pixmap button. It can contain other widgets to form even more complex structures.

4.2.1 RTK generic base widget class

RTK defines a base class for RTK widgets, the RTK generic base widget class. It is itself is derived from the RMP generic base widget class. It extends the functionality, implementing a set of methods and adding a set of attributes that are used by all RTK widget classes.

The RTK generic widget class is also an *abstract* class. It is not necessary to create a widget object of this class.

4.2.2 Widget *new* method

Every RTK widget implements a mandatory *new* method (see section 2.6.1, page 22). The *new* method takes care of the proper setup of the widget object. In the case of *complex* widgets it also creates additional widgets for internal use. The constructor returns a handle to the created widget.

4.2.3 Simple widgets

Simple widgets are the basic components in RTK. Some of them can be integrated directly into a user application while others are basically used to build more complex widgets. A *label* widget, for example, can be used in an application to display some text information. A *canvas* widget can be a box, or container for other widgets to group them in the user interface.

A button, however, rarely makes sense without a label. It *can* be used without a label, but the user might be confused about the purpose of such an element on the screen. The application should use a more complex button widget in this case.

4.2.4 Complex widgets

Complex widgets are built from other widgets. These other widgets can be *simple* widgets (which is usually the case) or other *complex* widgets. The constructor of the widget will take care of creating all contained widgets and nesting them into the widget structure. It also assigns default names for the nested subwidgets which allows the application to identify them (see section 2.1.2, page 15).

Regarding the resulting widget tree, there is no difference if an application uses a complex widget or builds the complex widget structure by itself. However, the widget constructor greatly simplifies the procedure of creating complex widgets.

4.2.5 Class hierarchy

RTK extends the generic functionality of the RMP generic widget class by introducing the RTK generic widget class. The RTK generic widget class adds some attributes and generic methods for supporting RTK specific tasks. Figure 14 shows the class tree for RTK widget classes.

4.2.6 Widget tree

Main window widgets are kept in a linked list inside the RMP core module. However, RMP does not cerate this list automatically. The RTK main window widget registers itself in the RMP widget list when it is created. As defined by the RMP generic widget class, every main window itself can contain other child widgets which again can contain child widgets themselves. RMP keeps track of these widgets by building a *widget tree* structure for every main window.

The user application uses the:

```
void rmp_add_widget(RMP_WIDGET parent, RMP_WIDGET child);
```

function to nest widgets.

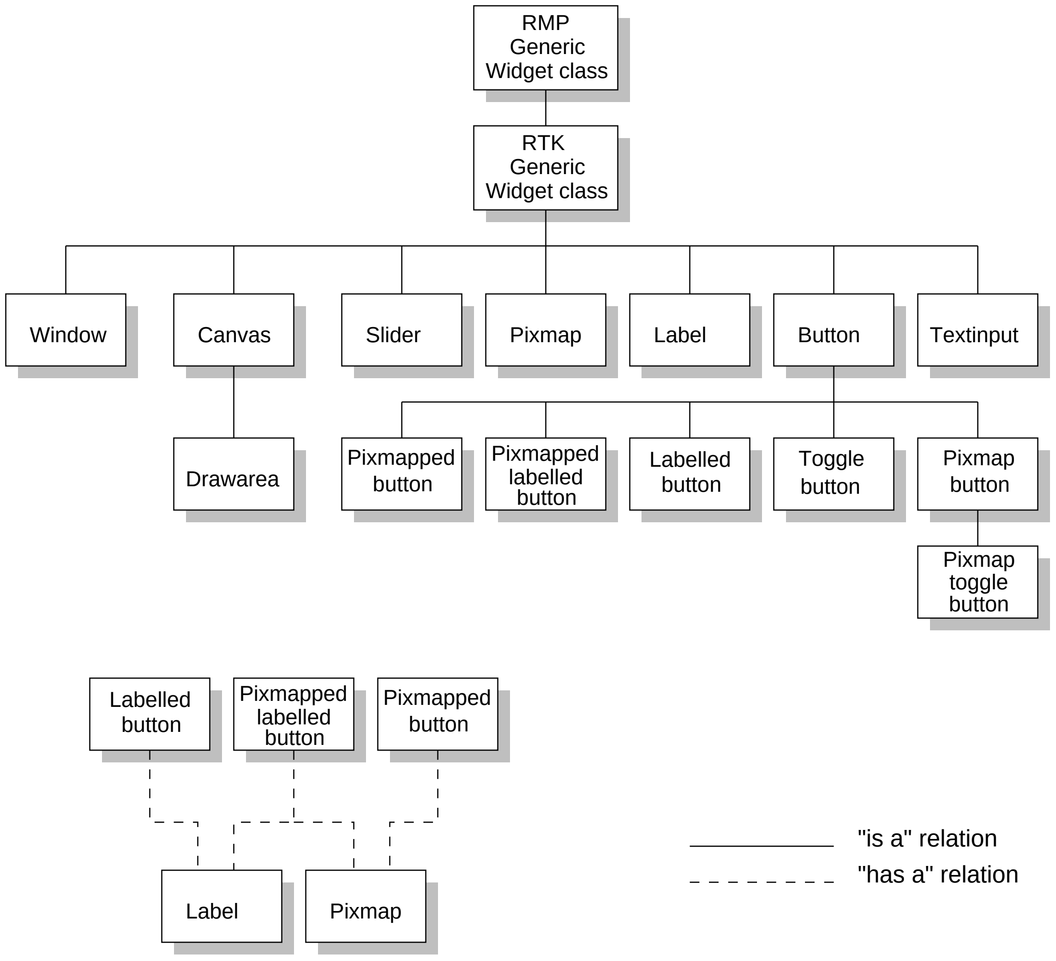


Figure 14: RTK widget class hierarchy

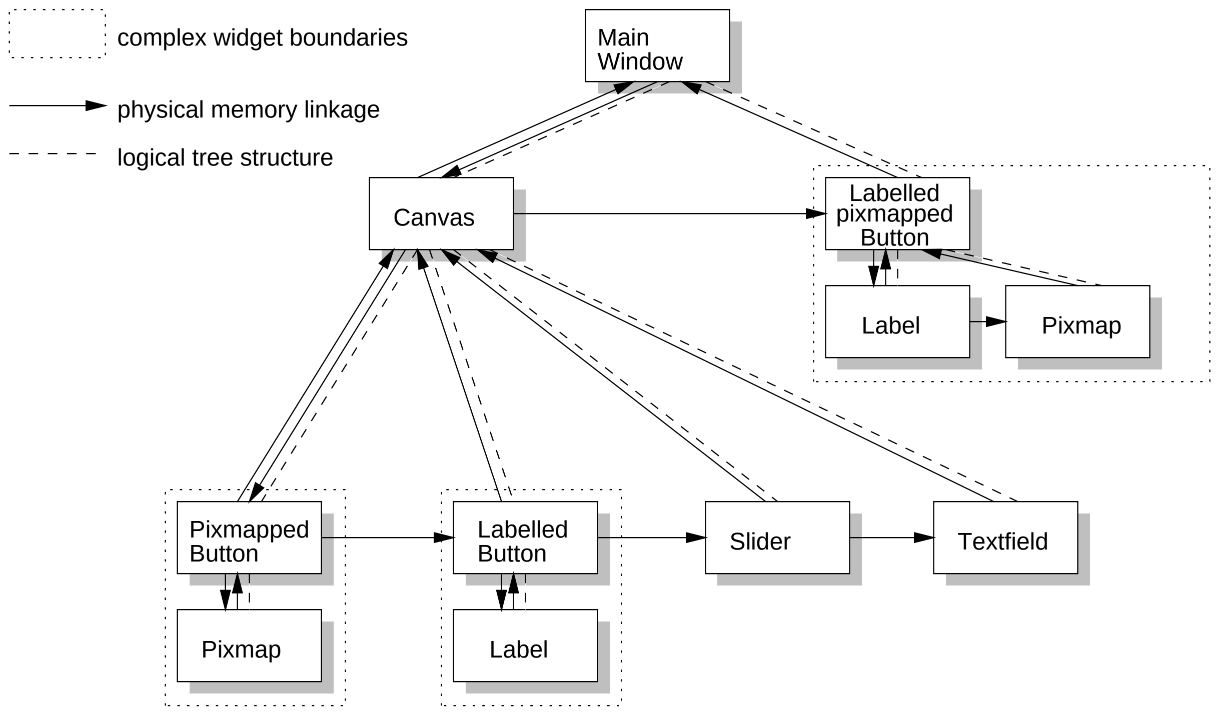


Figure 15: RTK application widget tree

4.2.7 The window widget

The window widget has a special rôle inside RTK. It is almost like other RTK widgets, so every function can be applied on it, but there are some points in which it differs from other RTK widgets:

- It is the only widget that has an associated drawable
- It has no parent
- It is the only widget that is actually drawn on the screen when the RTK *show* function is applied on it
- It is the only widget type that causes the window manager to draw a decoration around it

All windows that appear on the screen are created using window widgets.

4.2.8 Event handling in widgets

RTK widgets can handle certain GIA events by implementing a event handler method. If a GIA event (e.g. a mouse click) is received by RTK, the RTK core determines the widget that should receive the event (if any). The widget itself implements its behaviour and reacts according to the event by creating an *action* according to its event translation algorithm. These *actions* are usually sent to its own action handler. It can change its internal state and, for example, change its appearance.

The concept of dividing *events* and *actions* makes widgets extremely flexible. As well as from default behaviour to react to mouse events, widgets can be manipulated by any other action that is sent to them from any other source.

4.2.9 Position and gravity

When nesting widgets, it is possible to define the children widget's position within the parent widget. Usually the *window* widget of an application is the *root* widget and contains all child widgets that build the user interface. These child widgets like buttons, textfields or labels will be nested inside a *window* widget. The position of child widgets can be defined using

- absolute coordinates

or

- relative coordinates and gravity

When using relative coordinates the gravity can be a combination of

- top
- bottom
- left
- right
- center

gravity which causes a widget to snap into the desired direction. If concurrent gravities are combined they will be prioritized as shown in the list.

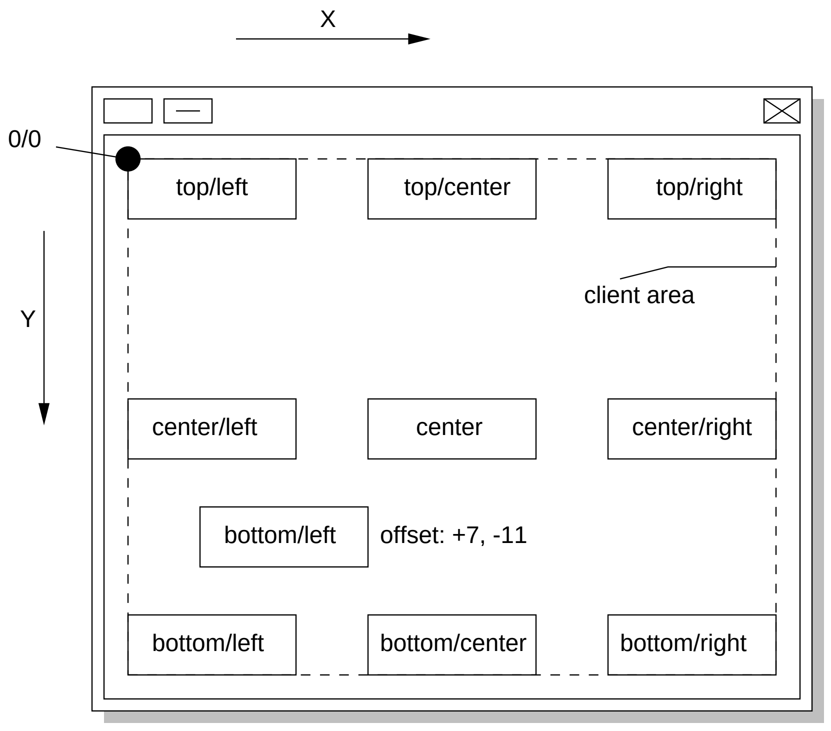


Figure 16: Widget gravity

When offsets are given in addition to the gravity then the position of the widget results out of the relative position *plus* the offset (see figure 16).

4.2.10 Focus

Events from the adapter layer that are handled by the RTK event handler will be forwarded to the *active* widget. To determine which widget is currently *active*, RTK sets the *focus* to a certain widget. When a widget gets *activated* by a mouse click event, it automatically gets the focus. It keeps the focus as long as no other widget gets *activated* by a mouse click. It loses the focus if

- another widget gets the focus

- a mouse click occurs in another desktop area

If a user, for example, wants to move a slider, it is not necessary that the finger follows exactly the shape of the slider on the screen. As the slider widget gets the focus on the mouse click, it will receive any mouse event as long as the mouse is moved, independent of the mouse's position, until the button is released. Assume a horizontal slider, the slider would receive *all* mouse events and coordinates but only process the *x* component of the mouse's coordinates regardless of its position on the screen.

The RTK core will translate mouse coordinates before sending events to the widget's event handler. The mouse coordinates will be relative to the origin of the widget. Coordinates can be negative. It is up to the widget to decide whether or not the coordinates make sense.

4.3 Widget abstraction

As with all toolkits in RMP, RTK uses the RMP widget handle to abstract its widgets. When a user application creates a RTK widget, it returns a handle of the type:

```
RMP_WIDGET
```

The handle is, per RMP definition, independent of the actual widget type and can be transparently used by a user application to reference the created widget.

4.4 RTK generic API

RTK provides the generic toolkit API described in section 3.7:

```
void rtk_init(void);
int  rtk_handler(ROME_MESSAGE * ev);
```

These generic functions implement the basic functionality of RTK within the standard RMP framework.

4.4.1 Create a widget

To create a widget the application simply uses the *new* macro of the desired widget. The macro is defined to:

```
RTK_WIDGETTYPE_NEW(optional params, const char *name);
```

The application can provide a name for the widget to identify it within the user interface or just give an empty string, if no name is required. The macro will call the *new* method of the appropriate widget which returns a RTK widget handle to identify the widget on future operations.

The list of optional parameters depends on the widget type. Some widgets allow additional parameters on creation to simplify the procedure. A labeled button, for example, can be created with one API call, providing its size and label text in the parameter list.

4.4.2 Delete a widget

Deleting a widget is one of the RMP core functions. RTK does not provide a *delete* function in its core, but every RTK widget implements a *delete* virtual method for the RMP core. Deleting a RTK widget is done by calling the *RMP delete* function:

```
void rmp_delete_widget(RMP_WIDGET widget);
```

giving the widget handle. This will delete the widget itself as well as *all nested* widgets. If this method were applied to the "main_window" widget in figure 3, all three widgets would be deleted. RMP will *only* delete the internal widget structure(s) by calling all their *delete* methods, but it will *not* update or remove any visible widgets on the screen.

4.5 RTK API extension

RTK widgets mostly represent *visible objects* on the screen. RTK extends its API to satisfy the needs of the extended functionality of the RTK toolkit itself and the extended capabilities and properties of its widgets. RTK provides additional functions to:

- Show a widget
- Set the position of a widget
- Set the orientation (gravity) of a widget

These methods can be applied on *all* RTK widgets. Other methods only apply to a certain subset of widgets or are only available for a particular widget.

4.5.1 Show a widget

To show a widget the application uses the RTK *show* function:

```
void rtk_show(RTK_WIDGET widget);
```

If the widget is a "*window*" widget, the window will be shown on the screen. RTK will cause GIA to copy the offscreen drawable buffer to video memory. If other widgets inside the window are already present in the drawable buffer, they will also appear on the screen.

If the widget is *not* a window widget, this function will only *draw* the widget into its window drawable, but *not* update or show it on the screen, unless the *auto update flag* of the widget is set. However, this feature should only be used when the window is already present on the screen. Otherwise individual widgets might appear on the screen without any context.

To make a a single *non window* widget appear on the screen, the RTK *update* function must be used.

4.5.2 Update a widget

Once a widget is drawn inside a window and the window is shown on the screen, it is usually necessary to change the widget's appearance. This is done by:

1. Drawing the new widget using the *show* function (draw in offscreen memory);
2. Updating the widget using one of the the *update* functions (to copy to video memory):

   ```
   void rtk_update_widget(RTK_WIDGET widget);
   ```

   or

   ```
   void rtk_update_widget_area(RTK_WIDGET widget, GIA_RECT * rect);
   ```

To increase performance the *update* function will only copy the area of the drawable buffer that is actually covered by the widget or the area of the widget that is covered by the defined rectangle.

Some widget methods cause *actions* to be sent to the widget. These *actions* usually cause a change in the appearance of the widget (e.g. text change in a label) and cause the widget to update itself. In these cases the *update* function needs not to be called by the user application.

4.5.3 Set the position of a widget

It is possible to change the position of a widget while it is present on the screen. To change the position there are two possibilities:

- change the orientation (or gravity):

  ```
  void rtk_set_orientation(RTK_WIDGET widget, uint gravity);
  ```

  The value for `gravity` is defined as a logical or of possible gravity values as described in section 4.2.9, page 40. When changing only gravity, the widget will keep its coordinates and use them as offsets for the new position;

- change the coordinates:

  ```
  void rtk_set_xy(RTK_WIDGET widget, int x, int y);
  ```

  When changing the coordinates only, the widget will keep its orientation and just move to the new coordinates, whether they are absolute or relative to a gravity.

5 STK - Sound ToolKit

In a multi-media environment sound plays an important rôle. An user application can use sound in many ways such as emphasizing graphical events on the screen or giving acoustical feedback on user actions.

For information kiosk systems, sound support is an effective way of presenting information to the user. It makes the use of such systems much more intuitive for inexperienced users.

On other systems, sound I/O might be the only way to communicate with the system at all. This could be systems which are operated by vision impaired users or in environments where it is not possible to use a screen.

Using sound widgets is a very convenient way to integrate audio into ROME applications. The versatile nature of the widget abstraction and the flexible widget–widget communication mechanism make it very easy to build complex and powerful multi-media applications.

For example, it is very easy to connect a *clicked*-action of an RTK widget with a *play*-action of a STK widget to cause a sound to be played when a button is clicked. All the framework needed to do this is already provided by the toolkits.

5.1 STK overview

The Sound ToolKit (STK) presents, much like RTK, a set of widgets to support sound I/O using a relatively simple API. STK is also a *shared library* located in the *toolkit layer*. It presents an API to the user application for managing and using sound objects, also called *widgets* as part of the RMP architecture.

STK again shows how flexible and powerful RMP is. Figure 17 shows an overview of the STK integration in the system. The comparison of Figure 17 and Figure 13 on page 36 shows that the basic concepts of integrating a toolkit into RMP are nearly the same for both toolkits. The only difference between STK and RTK integration is that STK is handling dataflow messages instead of events.

5.2 STK widget classes

STK implements a set of widget classes for creating STK widgets. These widget classes implement the basic functionality that is needed to play and record sound samples. Applications use STK widgets as *sound objects* which abstract sounds or sound samples at the level necessary to implement their audio capabilities.

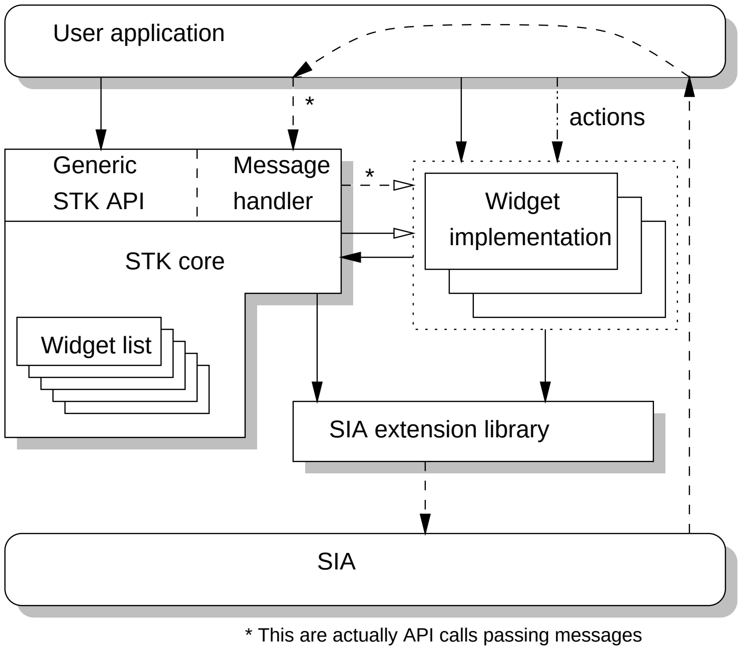


Figure 17: STK overview

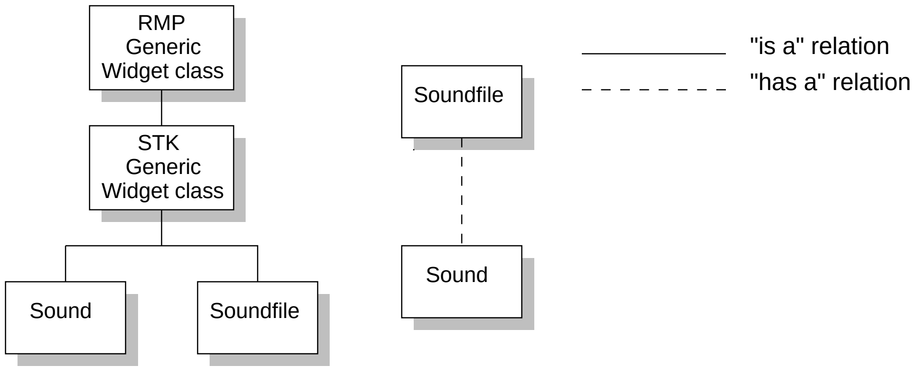


Figure 18: STK widget class hierarchy

5.2.1 Class hierarchy

STK extends the functionality of its widgets by introducing a *STK generic widget class*. The *STK generic widget class* is derived from the *RMP generic widget class* and provides a base class for all STK widget classes. It is the anchor for all widget classes in STK.

Figure 18 shows the STK class hierarchy which is relatively simple. Future widget classes are likely to be developed and to be integrated into the current structure. As shown in this figure, STK currently implements two widget classes:

- sound widget class
- sound file widget class

5.2.2 Sound widgets

Sound widgets are the basic components of the sound system. They provide the functionality to record and play back sound samples using widget internal memory buffers. Depending on the usage of the sound widget, it can use either its own memory buffer to store sound samples or use a buffer provided by the user application.

Sound widgets are volatile. As soon as the widget is deleted by the user application, the sound sample buffer will be deleted as well (unless the user application provided the buffer). Sound widgets are usually used for temporary purpose such as handling streamed audio.

5.2.3 Sound file widgets

Sound file widgets are always connected to a specific file (or URL). All their operations are associated with that file. They can be used to play back an existing sound file from an URL or to create a sound file and save it.

5.3 Widget abstraction

As with all toolkits in RMP, STK uses the RMP widget handle to abstract its widgets. When a user application creates a STK widget, it returns a handle of the type:

```
RMP_WIDGET
```

The handle is, per RMP definition, independent of the actual widget type and can be transparently used by a user application to reference the created widget.

5.4 STK event handler

When using STK an user application does not need to handle replied RMP messages. However, it *must* send all received messages to the STK message handler before handling any messages by itself. STK messages are tagged by the STK core before being sent so they can be recognized in the STK event handler.

STK's message handler will return a status:

```
RMP_HANDLED
```

or

```
RMP_NOT_HANDLED
```

to indicate the status of the operation.

5.5 STK generic API

STK widgets provide a set of generic methods that can be used on all STK widgets. These are the mandatory generic functions:

```
void stk_init(void);
int  stk_handler(ROME_MESSAGE *msg);
```

As with other toolkits, all RMP generic functions can be applied to STK widgets.

5.5.1 Create a widget

To create a widget the application simply uses the *new* macro of the desired widget. The application can provide a name for the widget to identify it within the user interface or just give an empty string, if no name is required. STK will return a STK widget handle to identify the widget on future operations. The *new* macros are defined as:

```
STK_WIDGETTYPE_NEW(optional params, const char *name);
```

The number and type of `optional params` depends on the actual STK widget type. The macro implicitly calls the appropriate widget *new* method and returns a RMP handle to the new widget.

5.5.2 Delete a widget

Deleting a STK widget is done by calling the *RMP delete* function:

```
void rmp_delete_widget(RMP_WIDGET widget);
```

giving the widget handle. The *delete* function will call the widget delete virtual method, so the widget can free all associated buffers or write any pending data back to disk.

5.6 STK user callback support

In many cases the user application needs to know when a sound sample has been played or there is an incoming audio data stream. As the RMP architecture supports the mechanism of user callbacks for its widgets, STK uses this feature to interact with the user application.

All STK widgets implement an action handler to support RMP widget–widget communication. Widget action handlers will forward the actions which they received to a user callback function if one has been defined.

As well as forwarding received actions to user callback functions, STK widgets create own actions to inform the user application about status changes within the widget. Currently STK widgets only create the:

```
STK_AC_READY
```

action to inform the user application that they entered an idle state and are able to receive new commands.

5.7 STK API extension

In addition to the RMP mandatory API, STK provides an extended set of functions for use with sound widgets. All these functions are implemented in the STK core and cause *actions* to be sent to the specified widgets, when they are called. This simplifies the API and makes the widget implementation more flexible.

5.7.1 Start audio playback

To start the playback of a widgets sound buffer, the application calls the:

```
void stk_play(RMP_WIDGET widget);
```

function. The widget will play back its associated sample buffer and stop when finished.

5.7.2 Stop audio playback

The playback of an audio sample can be stopped using the:

```
void stk_stop(RMP_WIDGET widget);
```

function. The widget will stop playback at the current position and the read pointer will be reset to the beginning of the audio sample buffer.

5.7.3 Pause audio playback

To pause the playback the application can use the:

```
    void stk_pause(RMP_WIDGET widget);
```

function. The pause function will stop the current playback but keep the read pointer at the current position. This emulates the behaviour of a pause button on a tape recorder.

5.7.4 Resume audio playback

When a widget is in pause mode, playback can be continued using the

```
    void stk_resume(RMP_WIDGET widget);
```

function. Playback will resume from the position to which the current read pointer points.

5.7.5 Start audio recording

Recording of sound samples is initiated by calling the:

```
    void stk_record(RMP_WIDGET widget);
```

function. The recording will continue until it is stopped or paused.

6 Future toolkits

The currently supported toolkits in RMP focus on *multimedia* capabilities and mostly provide *graphic* and *sound* support. But graphic and sound support are only two areas among many that are used in a multi-media oriented networking environment. Using the *widget* abstraction mechanism of RMP one might think of many other implementations such as:

- Network widgets
  Network widgets could provide a set of generic network operations to simplify data transport over a variety of different physical networking systems. They also could provide data encryption, QoS capabilities or network management features on a very abstract level.

- Video stream widgets
  Unlike the fairly static properties of GUI elements, video streams are characterized by large dataflows with timing and synchronization constraints. Rather than attempt to extend the RTK toolkit, video stream widgets could contribute to the already existing RTK by handling video streams of different types while interacting with network widgets. Video stream widgets could easily be integrated into existing applications or used to build video applications from scratch without disturbing existing GUI applications.

- Service widgets
  Service widgets could be used in active network systems, providing a certain subset of services that could be requested over a network. By using service widgets in combination with network widgets, active network elements could be easily built and integrated into existing networks.

The shown examples are only a few possibilities of extending RMP with new, innovative, flexible toolkits. The flexibility of RMP combined with the high performance architecture of ROME provides a solid base for fast, efficient, stable and reliable systems.

7 The Adapter layer

Within the RMP architecture, toolkits provide the most abstract representation of the system and its hardware. They present a hardware independent set of widgets to the user application layer, making development of portable user applications very easy and efficient.

Toolkits do not access the system hardware or the hardware drivers directly. They do not differentiate between devices on a low level. Instead they group hardware by functionality and provide widgets to represent a certain functionality. For example, STK provides a widget for managing sound samples. The widget *only* represents the *sound I/O* as a hardware functionality, it has *no* relation to any physically existing sound support hardware.

Toolkit implementations are based on another level of hardware abstraction which is provided by *adapter processes* in the *adapter layer*.

7.1 Adapter processes

The *adapter layer* provides a level of hardware abstraction on which the RMP toolkits is based. The abstraction is implemented as a set of *adapter processes* which are located in the *adapter layer*. Adapter processes can be rather simple or very complex, depending on the hardware type and level of abstraction they provide. They are not necessarily bound to *one* specific device type, they can implement support for different devices.

7.1.1 Minimal adapter processes

The basic task of an adapter process is to provide a connection between the toolkit layer and the device driver layer. In the simplest scenario this means that the adapter process receives and forwards messages from the toolkit layer and to the device driver processes and vice versa. This behaviour requires a simple multiplexing functionality which is implicitly provided by the ROME message and event queue mechanism.

7.1.2 Complex adapter processes

It is possible to put much more functionality into an adapter process. This makes sense if the adapter process abstracts a complex type of hardware or a set of hardware components. As an adapter process is a ROME process like any other ROME process in the system, it can be used to implement complex algorithms at a relatively low level in the system. For the upper layer the functionality can be provided in form of APIs or control messages.

For example the GIA adapter process implements the complete behaviour of a desktop and a window manager by taking control over all graphical requests and input device messages and events. GIA is by far the most complex adapter process currently integrated into RMP.

The example of GIA show that adapter processes can provide an efficient control mechanism within an RMP based system.

7.1.3 Adapter process interface

Communication between adapter processes and *lower layer* device driver processes *always* happens through *messages* or *events*. Adapter processes can implement a *queue handler* for ROME messages and events just like any other ROME process. The message system provides all advantages of the ROME message mechanism such as message priorities and multiplexing.

7.1.4 Adapter extension libraries

To simplify the communication between a user process (or toolkit) and an adapter process, each adapter process provides a *extension library*. Extension libraries provide a shared library API for the user process and translate the calls into messages that can be understood by the adapter process. This mechanism has several advantages:

- For the user application it is easier to use an API than create and send messages which makes application development faster;
- The ROME message mechanism automatically takes care of sequencing requests and avoids resource conflicts;

- The ROME message priority mechanism allows to define different adapter access priorities for different processes;
- As messages can be sent over a network, the user application and the display system *do not* necessarily have to reside on the same hardware. This makes it possible to transparently run distributed applications over a network.

Extension libraries combine the advantages of having a shared library API with the event-processing and dataflow capabilities associated with the ROME process and message mechanism.

However, one must consider that for sending messages over a network the whole message including any payload data might have to be copied at the sender *and* at the receiver location. This counteracts the ROME *zero copy philosophy* and should therefore not be the common case.

7.2 Virtual devices

One task of adapter processes is to provide a level of hardware abstraction. Adapter processes communicate with the device drivers of the system and provide a more abstract interface to the upper layer. The abstraction is achieved by using methods like *data encapsulation*, *data mapping* and *data translation* (see section 1.4, page 12).

With adapter processes it is possible to abstract *a set of devices* to a *virtual device*. For example, it does not matter if the input device is a *mous*e, a *touchpad* or even a *light pen*. The adapter process will translate the data and present a generic *pointing device* interface to the upper layer. Toolkits can implement a functionality based on the properties of that *virtual pointing device* rather than implementing functionality for each conceivable physical device.

If new devices are added to an existing system, it is only necessary to implement a new abstraction algorithm into the adapter process. The rest of the system does not have to be changed.

7.3 Device properties mapping

The principle of having virtual devices allows adapter processes to *map* certain physical hardware functions to *logical functions*. A virtual device has a certain set of generic characteristics. For example, a virtual pointing device currently supports a set of coordinates and the device state. Because different hardware of a certain set of devices has different physical properties such as different buttons and switches of a mouse or a light pen, the adapter process defines a *mapping* from the hardware properties to the virtual device properties.

This mechanism makes it possible to define completely new input methods while the function of a virtual device can be emulated by almost any available hardware. This makes the software not only hardware independent, but also device *type* independent.

7.4 Messages and events

The mechanism of virtual devices and device properties mapping requires that the adapter process defines a set of new messages and events for the system. These messages and events are used to communicate with the rest of the system. Adapter process implementations define both messages for lower level device drivers and messages for upper level processes and toolkits or other adapter processes. By defining the types of the messages and events adapter processes enable the other components to fit into the environment.

For example, an adapter process defines the messages and events for the virtual pointing device. To the upper layers it presents a message containing a set of coordinates and the current state. For the lower level device driver process an event containing a set of coordinates and a few opaque fields representing physical device buttons and switches is provided. Device driver processes use the pre-defined event and fill it with the data read from the device while the mapping algorithm in the adapter process maps it into the virtual pointing device message and passes it up.

This scheme allows different pointing devices to work with a toolkit. For example a touch-screen will generate absolute coordinates, while a mouse indicates only relative movement. The representations are unified within the adapter layer.

7.5 Control mechanisms

7.5.1 Dataflow control

All data from and to devices passes through the adapter process. This makes an adapter process the perfect place to take control over dataflows in RMP. Dataflow control can happen in different ways, depending on the data type and the requirements of the system. Adapter processes allow the dataflow algorithms to be adapted to the hardware requirements in a very flexible way.

7.5.2 Multiplexing and demultiplexing

An adapter processes can work together with many different device drivers. From the upper layers, many different user processes can use toolkits to access the hardware through the adapter process. One of its tasks is to multiplex and de-multiplex the requests from the different user applications to the different hardware device drivers. The complexity of this task depends on the number of devices supported by the adapter process, the individual device's multiplexing capabilities and on how much other functionality is integrated into the adapter process itself.

7.5.3 Resource conflicts

By managing the dataflows in the system, adapter processes avoid resource conflicts at a low level. For example, if two user applications want to access the same hardware at the same time, the adapter process can resolve that conflict. How the conflict is resolved, is defined by the adapter process policy. It might make sense to apply a *first come, first served* policy or to serve the request with the *highest priority* first. In some cases it might be possible to *merge* to requests. For example, if two applications each want to play a sound sample on the system's sound chip, the adapter process could merge the two samples at run-time and by doing so play them simultaneously.

8 GIA - Graphical Interface Adapter

The *Graphical Interface Adapter* (GIA) process is located in the *adapter layer*. It implements the standard adapter functionality of multiplexing and de-multiplexing messages and events from user applications and hardware device drivers and preventing resource conflicts in the system. It includes support for multiple *input devices* like a keyboard, mouse or touchpad as well as *output devices*. The output device currently supported are the WebPanel LCD screen and the Number9 graphics card. Input devices are a touchscreen and a keyboard over a serial line as well as a serial mouse which has been added recently.

Additionally, GIA implements the concept of *drawables*. Drawables basically are containers for graphical elements that can be used by user applications (see section 8.3, page 53). Drawable buffers will be managed by GIA and can be manipulated using the GIA API. This abstraction of the graphics hardware makes application programming very generic. Usually a toolkit like RTK sits on top of GIA, utilizing drawables for drawing its windows and widgets.

Keeping drawable buffers within GIA results in an improved performance. Buffers are kept as near to the graphics device driver as possible, reducing data path lengths in the system. Furthermore, as GIA has total control over the drawable buffers it is possible to implement very effective algorithms for clipping and repainting areas on the screen.

GIA also supports the use of an optional *window manager*. The standard window manager (RWM) is implemented as a shared library extension to GIA and provides standard windowing functionality like *moving* and *resizing* windows on the desktop.

8.1 GIA overview

GIA supports the management of drawable areas only in a very elementary way. It does not provide frames, buttons or menus. This functionality is implemented by toolkits like RTK (see section 4, page 35) which sit on top of GIA. The main tasks and key features of GIA are:

- Managing and manipulating drawables;

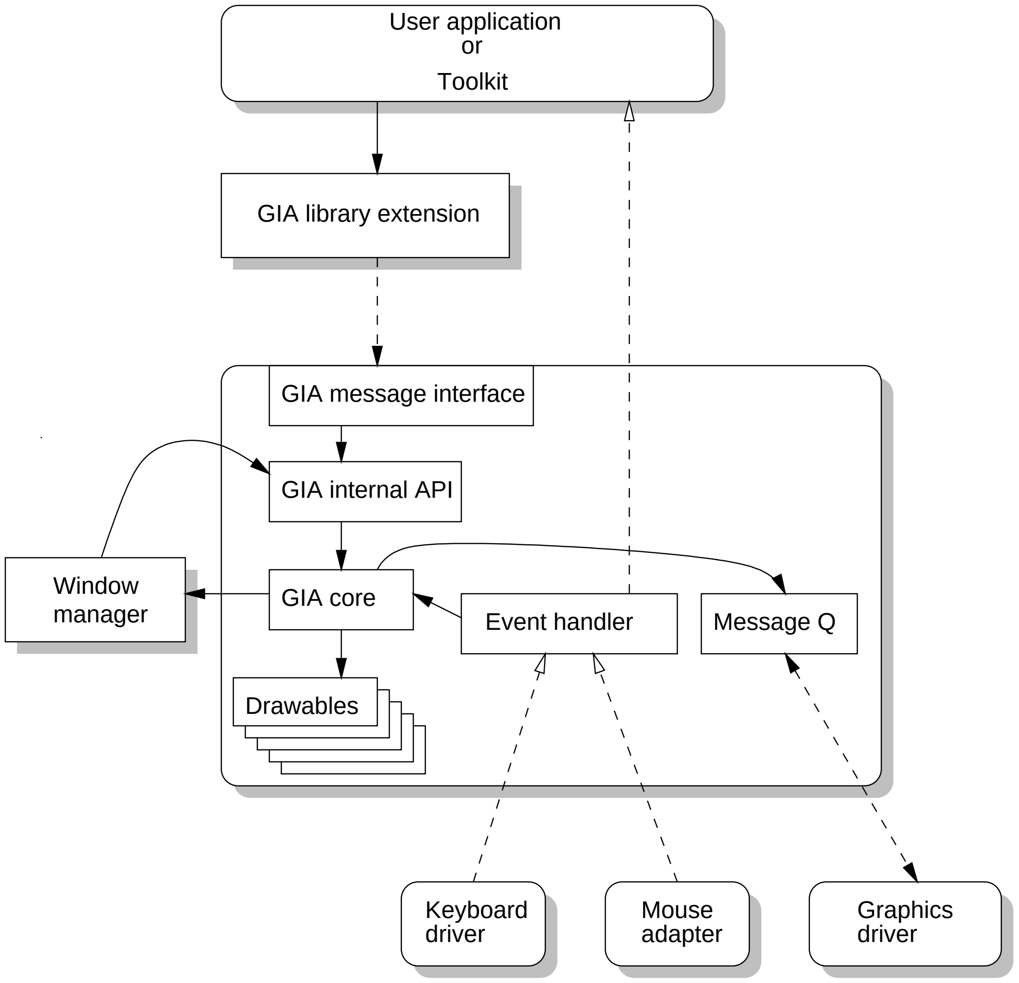


Figure 19: GIA overview

- Keeping track of all drawables and their stacking order on the desktop;
- Interaction with the low level graphics driver;
- Painting the drawables onto the screen;
- Receiving and processing events from mouse, touchpad, keyboard or other input devices;
- Generating high level mouse, keyboard and drawable events for RTK;
- Interaction with an optional window manager.

8.2 GIA configuration

As GIA is a very flexible, versatile process it must be configured for its input and output devices. This is done using the ROME System Manager, where entries can either be made in the system configuration file or placed as static entries in the application build file. GIA is then setup at boot time using a set of command entries, for example:

```
[boot] :gia:graphics lcd
[boot] :gia:mouse mouse
```

After receiving a configuration command, GIA tries to open the specified input or output device.

8.3 GIA drawables

For abstracting output devices GIA uses the concept of drawables. Drawables are offscreen buffers which represent a potential area on the screen and can be manipulated by GIA API calls. Drawables are completely encapsulated into GIA and are referenced using handles.

A drawable usually represents a window on the screen, but in general it is just a *generic canvas*. Changing the drawable content is always done offscreen, thus only changing the GIA internal drawable buffer. To make a change in the drawable visible, the drawable has to be updated on the screen. It is also possible to update or redraw only certain areas of the drawable to increase performance. The concept of GIA drawables provides several advantages:

- GIA is able to control all graphic output, as all screen access have to occur through drawables;
- Keeping all drawable buffers inside GIA makes it possible to implement effective algorithms for updating screen areas. This code can be implemented in the GIA core;
- Events from device drivers that only affect drawables can be processed within the GIA core or event handler, without interference of a user application process (for example, moving a window);
- User applications *do not* have to repaint parts of their user interface once a hidden area of the window gets exposed.

8.3.1 Drawable handles

Drawables are always referenced by handles. Handles are unique identifiers that must be used for every operation on a drawable using the GIA API. They are returned to the caller when a new drawable is requested and are of the type:

```
GIA_DRAWABLE
```

Drawable handles are validated on any operation through the GIA API. This prevents invalid handles being processed by the GIA core.

8.3.2 Drawable types

GIA distinguishes between several types of drawables. This separation makes it possible to implement different behaviour or processing algorithms for the drawables depending on their type and purpose. For example, a background drawable clearly has a different intended purpose and behaviour than a *sprite* drawable.

The user application (or caller, in general) has to specify the type of the desired drawable. All drawable types are specified in `gia.h`:

`GIA_TYPE_MAIN_WINDOW` Main window drawables represent the standard main windows of applications.

`GIA_TYPE_BACKGROUND` There is usually only one background drawable (per desktop, if there are many) existing in the system. The background drawable will always be stacked lowest in the drawable stacking order. This way, the generic redraw function will always put it *behind* all other drawables.

`GIA_TYPE_MOUSE` On most systems a pointing device will be attached. To represent the position of the device on the screen, a mouse drawable is used. Mouse drawables behave differently when other drawables are updated on the screen. When the updated area overlaps with the mouse drawable, the mouse drawable must be switched off during the paint operation and then redrawn.

`GIA_TYPE_OTHER` other drawables, better known as sprites, are used for small objects like tooltips, icons or small pull down menus. When these drawables are created, their current background on the screen will be captured and repainted when they move or disappear. However, the use of this drawable type is only recommended for relatively small drawables.

`GIA_TYPE_OTHER_MASK` This type basically represents the same type as `GIA_TYPE_OTHER`, however the drawable can contain a *transparent* color. Otherwise it can be used and behaves like the `GIA_TYPE_OTHER` drawables.

8.3.3 Color depth, color translation

When a drawable is requested from GIA, it creates it based on the device's color depth. To compute the amount of memory needed for the drawable, GIA needs to know about the graphics hardware capabilities and the current graphics mode. The graphics device driver must respond to a message to request the current graphics device capabilities which contains the size and the color depth of the screen. GIA uses this message to determine the amount of memory needed. This saves resources for systems with less color depth.

However, to simplify things, applications always work with 24 bit color depth. The application can simply assume that the hardware supports 24 bit color depth. This way no additional code has to be implemented in the user application. For example, the XPM library also only supports image types with 24 bit color depth.

Internally all GIA functions process colors with 24 bit, so GIA transforms the color codes if the hardware does not support 24 bit color depth. GIA requests the color format from the driver and uses color database library (See section 14.3, page 73) macros for translating color codes into the appropriate format.

As all GIA the development focused on the WebPanel, GIA currently only supports translation to 16 bit, 565 RGB color format. Other color formats will be added in future, if new graphics hardware requires it.

8.3.4 Drawable graphic context

Usually, drawables represent windows on the desktop. When a user application wants to draw some graphical elements into the drawable, it uses the drawing functions provided by GIA. All drawing coordinates inside drawables are relative to the drawable origin. If the application wanted to address an area within the drawable, thus introducing a canvas inside the drawable, it always would have to compute the coordinates of the canvas relative to the drawable, checking for valid coordinate pairs.

To simplify things, GIA introduces the mechanism of *graphic contexts*. Graphic contexts define rectangular areas within drawables, defined by an relative origin within the drawable and a size. they can be understood as *drawing windows* within a drawable. The application can specify a graphic context to draw inside that window. GIA takes care of coordinate checking and simply discards any request outside the graphic context area. This makes it possible to use, for example, negative coordinates to draw a line, as only the *valid* part of the line is visible in the drawable.

The mechanism of graphic contexts is heavily used by the RTK toolkit. It uses graphic contexts on a per-widget basis which greatly simplifies the handling of the graphical side of RTK widgets.

8.4 Managing drawables

As GIA is an adapter process, usually a toolkit is sitting on top of it. User applications mostly are built on top of toolkits and therefore use the toolkit's features, while the toolkit accesses the GIA API. Only in some cases do user applications directly use the GIA API themselves. In the following the term *user application* always refers to a *user application + toolkit combination* where the toolkit actually accesses the GIA API.

To manage drawables GIA provides a set of interface functions. They are used to:

- Create drawables
- Delete drawables
- Map drawables to the screen (make them visible)
- Unmap drawables (this will NOT delete the drawables, just remove them from the screen)
- Paint drawables
- Repaint drawables
- Move drawables to another position (mostly used by a window manager)
- Resize drawables

8.4.1 Create drawables

To create a new drawable the user application uses the:

```
GIA_DRAWABLE gia_new_drawable(GIA_DRAWABLE_ATTR *attr);
```

function. The function will return a handle to the new drawable or `NULL`, if the function failed. The `attr` points to an attributes structure which is defined as:

```
typedef struct gia_drawable_attr_d
{
    int type;            /* drawable type */
    char *name;          /* optional name */
    GIA_RECT rect;       /* size and position */
    GIA_RECT child_rect; /* rectangle of child area */
    int back_buff;       /* flag for background buffering */
    char *image_buf;     /* alternative image buffer */
}
GIA_DRAWABLE_ATTR;
```

and contains the properties of the requested drawable.

8.4.2 Delete drawables

When a drawable is no longer needed by an application, the application can delete the drawable. This is done using the:

```
int gia_delete_drawable(GIA_DRAWABLE drawable);
```

function. The drawable will be deleted from the GIA drawable list and also removed from the desktop (unmapped). All associated memory buffers will be freed (unless they were provided by the user application).

The function returns 0 on success, -1 if an error occurred.

8.4.3 Map drawables

To make a drawable visible on the desktop it must be *mapped*. When a drawable is mapped, its drawable buffer will be copied to the video memory by sending a request to the graphics device driver. This is achieved by the:

```
int gia_map_drawable(GIA_DRAWABLE drawable);
```

function. The *map* function must not be confused with the *paint* or *repaint* function. Although they seems to have similar functionality, the *map* function additionally invokes *window manager* functions which causes the window manager to create decorations for the drawable. In general a drawable may be mapped only once but repainted many times.

The function returns 0 on success, -1 if an error occurred.

8.4.4 Unmap drawables

When a window is temporarily not used by the application it can be *unmapped*. Unmapping a drawable only removes it from the visible screen, it does not delete any associated structure or buffers. The unmap function is defined as:

```
int gia_unmap_drawable(GIA_DRAWABLE drawable);
```

The function returns 0 on success, -1 if an error occurred. The *map/unmap* combination may be used to implement *pop-up* style windows, or menus.

8.4.5 Paint drawables

To make a drawable visible on the screen, the:

```
int gia_paint_drawable(GIA_DRAWABLE drawable);
```

function must be used. It is usually not necessary to call the *paint* function, as the *map* function implicitly calls it when a drawable is mapped onto the screen. When parts of the window are covered by other windows and exposed again, GIA internally takes care of *painting* and *repainting* drawables.

However in those cases where the user application needs to update a drawable explicitly the *paint* function must be used.

The function returns 0 on success, -1 if an error occurred.

8.4.6 Repaint drawables

On the surface, there may seem to be no difference between the *paint* and the *repaint* function. However, there is one, it is subtle, but important. The *paint* function *redraws* the *background* of a drawable if drawable moved to another position. The *repaint* function:

```
int gia_repaint_drawable(GIA_DRAWABLE drawable);
```

does not which reduces the internal processing requirements since no checks for drawable movements are performed. Thus, the repaint function should only be used when the drawable contents have to be updated, but the drawable position did not change.

The function returns 0 on success, -1 if an error occurred.

8.4.7 Moving drawables

Because the *move* function is integrated into the GIA core, GIA has complete control over all necessary steps to move a drawable:

1. Remove the drawable form the screen (*unmap*)
2. Repaint the previously-obscured area behind the drawable (done by *unmap*)

3. Set the new position of the drawable in the drawable structure
4. Paint the drawable (*paint*)

Usually the *move* function:

```
int gia_move_drawable(GIA_DRAWABLE drawable, int x, int y);
```

is called by the window manager, although it could be called by any application if it wants to force a position for its window.

At this point the concept of drawables shows its strengths. As all window (or drawable) contents are already defined and do not change during a move, all repaints can be done using the GIA internal drawable buffers. there is no need to interact with applications to request new data.

8.4.8 Resize drawables

Drawables usually represent windows on the screen. During the lifetime of a window it might be necessary to change its size, either by a user request or due to changes of its content. For example, if a window is used for displaying pictures, it might be useful to adapt the window size to the individual picture's size. this could be caused by the view-application which sets the window size or by a user request, manually resizing the window frame.

User requests are processed by the ROME window manager, as the user drags the corner of the frame of the window. Application requests are processed inside an user application and passed to the GIA. either way they result in the call of the:

```
void gia_resize_drawable(GIA_DRAWABLE drawable, int dx, int dy);
```

function which resizes the desired drawable in GIA. GIA takes care to free and allocate drawable associated memory buffers, if no user buffers have been defined for that drawable.

8.5 Manipulating drawables

Once a new drawable is created, the user application usually needs to manipulate it. Drawable contents can be modified by using functions to:

- Draw single pixels
- Draw lines
- Draw boxes
- Draw frames[6]
- Put images (e.g. created by the XPM library)
- Clear drawables

With *any* operation on a drawable the application *must* provide the *graphic context* of the area it wants to manipulate. An application can calculate the *graphic context* of the area it wants to manipulate or obtain the *default graphic context* for the drawable by using the:

```
GIA_DRAWABLE_GC gia_default_gc(GIA_DRAWABLE drawable);
```

function. The function returns a graphic context that covers the *whole client area* of the drawable.

[6]A frame is a box with a defined line width

8.5.1 Draw single pixels

It is possible to set and reset single pixels inside a drawable. This might be useful to draw simple graphs. The application uses the:

```
void gia_plot(GIA_DRAWABLE drawable,
              GIA_DRAWABLE_GC gc,
              int x, int y, uint col);
```

functions to set and reset pixels. Resetting pixels is done by setting pixels with the same color as the background of the drawable. Note that *x* and *y* are relative to the *graphic context*, not to the origin of the drawable.

8.5.2 Draw lines

GIA provides the:

```
int gia_draw_line(GIA_DRAWABLE drawable,
                  GIA_DRAWABLE_GC gc,
                  int x1, int y1, int x2, int y2, uint col);
```

function to draw single lines of width *1 pixel* into a graphic context. The function returns 0 on success, -1 if an error occurred.

8.5.3 Draw boxes

Drawing rectangular boxes is done through the:

```
int gia_draw_box(GIA_DRAWABLE drawable,
                 GIA_DRAWABLE_GC gc,
                 int x, int y, int dx, int dy,
                 uint col, int fill);
```

function. The `fill` flag defines if the box should be filled or only a frame should be drawn. The function returns 0 on success, -1 if an error occurred.

8.5.4 Draw frames

The *frame* function:

```
int gia_draw_frame(GIA_DRAWABLE drawable,
                   GIA_DRAWABLE_GC gc,
                   uint width, uint col);
```

draws a frame with a specified width around the specified drawable graphic context. This function is mainly used internally by the window manager. The function returns 0 on success, -1 if an error occurred.

8.5.5 Put images

In order to put small graphical elements like icons and small pictures into a window the:

```
int gia_put_image(GIA_DRAWABLE drawable,
                  GIA_DRAWABLE_GC gc,
                  const char *image,
                  int x, int y, int dx, int dy,
                  int mask);
```

function is provided. The image data must be provided in form of a memory buffer. Usually such an image is created using the XPM library. The mask parameter defines if the image should be treated as a *transparent* image. The *put image* function returns 0 on success, -1 if an error occurred.

8.5.6 Clear drawables

To remove all contents from a drawable, the application uses the:

```
int gia_clear_drawable(GIA_DRAWABLE drawable,
                       GIA_DRAWABLE_GC gc, int col);
```

function. The function clears the whole drawable area to the specified color. The function returns 0 on success, -1 if an error occurred.

8.6 GIA strategies

When running multiple applications with many windows on a single desktop, the GIA layer ensures a consistent desktop appearance. Window contents can overlap, must be repainted or moved around on the visible screen. New windows can appear, or older ones disappear from the screen. For all of these cases it is necessary to repaint certain areas of the screen to always represent the current state of the working desktop. GIA implements several strategies to achieve this goal.

8.6.1 Background buffering

Background buffering is relatively simple. When using background buffering, GIA captures the background for a drawable and repaints the background, when the drawable is moved on the screen or if it is deleted. However, this method has some disadvantages:

- The method is only efficient for small drawables, because a buffer for the whole background of the drawable must be allocated and the whole background must be repainted every time which might result in an extreme video I/O.
- The drawable background buffer has to be refreshed, every time the background of the drawable changes. Otherwise the drawable would remember the background before the change and repaint it, when it is moved or hidden thus undoing any changes in the background.

GIA keeps track of such drawables and hides and unhides them automatically, if the background beneath them gets changed.

8.6.2 Background repainting

Repainting the background is more difficult than background buffering. Every time a drawable is moved on or deleted from the screen, an area on the screen will be exposed. The contents of drawables inside this area has to be repainted by GIA.

When moving or deleting a drawable GIA determines the rectangle of the exposed area. It builds a list of all drawables that overlap with that area while also considering the drawable *stacking* on the desktop. For every of these drawable it defines a set of rectangles that have to be repainted for that particular drawable and repaints it (see figure 20).

This method is desirable for larger drawables where background buffering would use too much memory and keeping track of changes would be too processor-intensive.

8.6.3 Clipping

A common situation is that areas of drawables are hidden by other drawables on the screen. When partially hidden drawables have to be updated, GIA determines the actual visible parts of the drawable and only updates these areas.

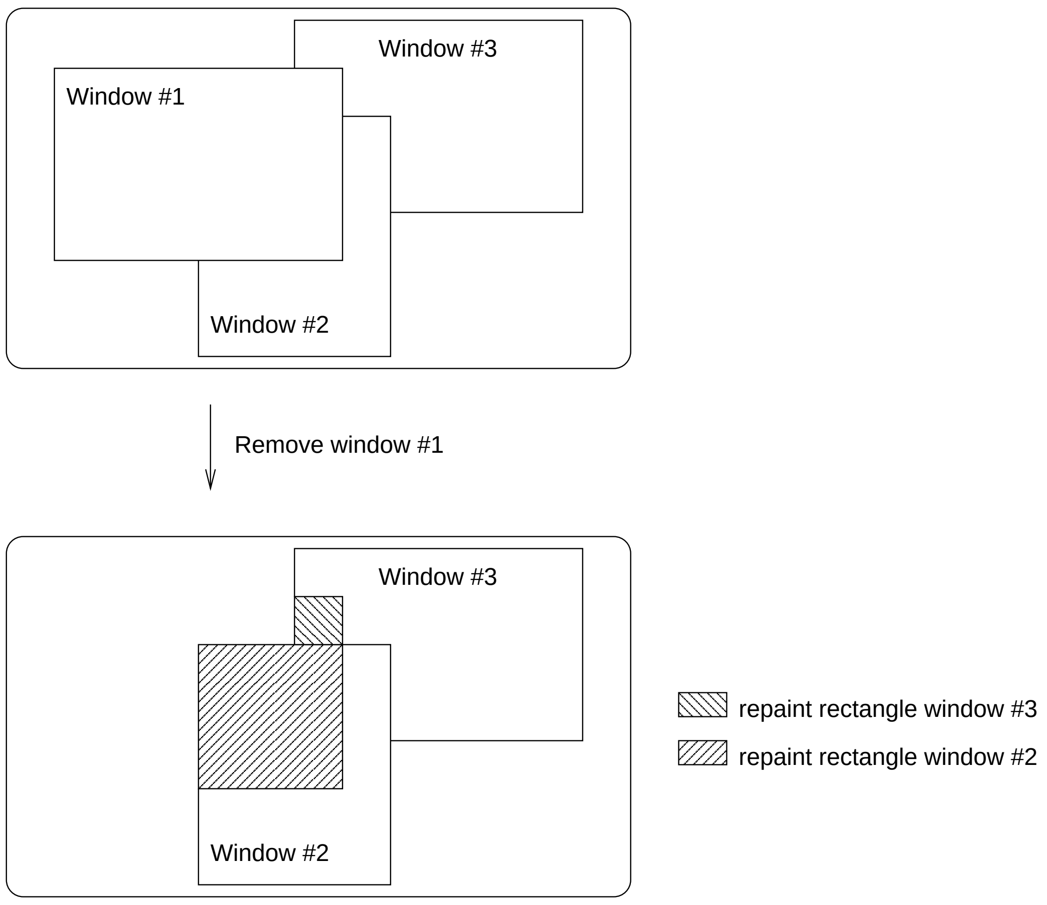


Figure 20: GIA background repainting

8.6.4 External drawable buffers

If an application requests a drawable, it can decide if the drawable buffer should be provided by the video driver, or if the application has its own buffer. This is useful, if the application for example receives video data it wants to display. Instead of copying the data from the video buffer into the drawable buffer it could directly provide the drawable with the video buffer.

If an application provides its own drawable buffer it also has to care to provide it in the appropriate format according to the graphics device capabilities. This option is only available if the graphic device is located on the local system.

8.7 GIA window manager interaction

In order to provide drawables with decorations and to allow them to be moved around the screen, to iconify them, to resize them or to rearrange them, GIA interacts with a window manager (see section 10, page 65). The window manager is a extension to GIA in form of a shared library. For events like mouse or keyboard actions, specific functions of the window manager extension library are called. This makes it possible for the window manager to position drawables on the screen or draw a decoration for window drawables.

8.8 Fonts

GIA supports Bitmap Distribution Format fonts (BDF). BDF is a format that is also widely used in UNIX and UNIX-like operating systems and font files are freely available on the web.

8.8.1 Font path configuration

The location of the BDF font files must be configured using the System Manager database. This can be done by adding an entry in the system configuration file or the static configuration list in the build file. The configuration entry must be in the *gia* section and specify the font path:

```
[gia] fontpath:=dos:/c/fonts
```

If the fontpath variable is not defined in the configuration database GIA will use a default value. If no font files can be found GIA will proceed without font support.

GIA will not fall back to *compiled in* fonts, as the option for compiled in fonts has to be set at compile time.

8.8.2 Font file support

Usually GIA loads BDF fonts from a disk when they are needed in the system. This saves resources as only fonts that are really used are kept in system memory. This allows a large variety of fonts to be stored on disk without wasting system resources. However, as one can not always assume that the target configuration contains a filing system, font file support must be explicitly enabled by using the option:

```
Option GIA_FONTFILE_SUPPORT
```

in the application build file. With this option enabled, GIA will scan the disk for available font files at boot time and build a small database with the *font names* and the associated *file names*. The database is used when GIA checks if a font is available and which file contains the font data.

8.8.3 Static fonts

If the system does not support a filing system, it is possible to compile static fonts in to the GIA code. If the:

```
Option GIA_FONTFILE_SUPPORT
```

option is *not* defined in the system build file, GIA will compile a generic terminal font in to the code.

8.8.4 Handles

Fonts are managed via font handles. If an application wants to use a font it has to request a handle for the font from GIA. Font handles are later used for referencing the font using GIA interface functions. The font handle type is defined as:

```
GIA_FONT
```

This mechanism works like the handle mechanism of toolkits.

8.8.5 Names

On startup GIA scans the entire font directory for valid font files and creates a mapping list of font names and filenames. Font names are structured in the following way:

```
family-size-weight-slant
```

for example:

```
Helvetica-18-Bold-R
```

Font names are created based on entries in the BDF font files on disk. GIA will map each font name to the related filename to be able to load them upon request. Fonts are always referenced by their font name rather than their filename. Applications must use the font name format. This makes the font handling independent of the actual filing system. Applications can use the same font names on *any* system regardless of the underlying infrastructure.

8.8.6 Caching

To save resources and shorten system boot time GIA only loads the font descriptions rather than complete font data. When a specific font is needed in the system, GIA loads the font dynamically into memory. Subsequent requests for the same font will result in accessing the font bitmap data previously cached in system memory.

8.9 Mouse adapter

GIA abstracts several devices including the pointing device. However, in the case of the pointing device GIA does not implement the abstraction algorithm itself. GIA does not receive events directly from pointing device drivers, it receives such events from the mouse adapter process. The mouse adapter is a separate adapter process located in the adapter layer, and takes care of a low level abstraction of different pointing device classes.

8.9.1 Device abstraction

The mouse adapter abstracts all attached pointing devices to a virtual pointing device. The virtual pointing device always delivers absolute screen coordinates and the state of a button.

If the pointing device already delivers absolute coordinates, the mouse adapter simply maps them into its own event and sends them to GIA. If the pointing device delivers coordinate deltas, the mouse adapter computes the resulting absolute coordinates before sending events to GIA.

8.9.2 Mouse control

The mouse adapter process defines the position of the mouse cursor on the screen. As it has to calculate the new position of the cursor, when it receives coordinate deltas, it keeps track about the current position of the mouse cursor.

An external process might want to change the position of the mouse. For example, the window manager might want to set the position of the mouse to the center of a newly created window. The mouse adapter process supports a command to set the position of the mouse cursor which can be sent from any other process.

8.9.3 Configuration

At boot time the mouse adapter receives *two* commands from the system manager:

- a *device* command, which defines the pointing device process and if the device reports absolute coordinates or coordinate deltas;
- a *screen* command, which defines the size of the screen. The *screen* command is only needed, if the pointing device reports coordinate deltas.

After this configuration the mouse adapter is able to receive and process mouse events from the device driver.

For example, the system manager database could contain following entries:

```
[boot]:
   :mouse:device serial_mouse 1
   :mouse:screen 640 480
```

for setting up a serial mouse on a 640x480 screen.

9 SIA- Sound Interface Adapter

Unlike the GIA, the *Sound Interface Adapter* process is a relatively simple process. Located in the *adapter layer* it provides abstractions of the sound hardware in the system by defining a generic sound sample structure in its external header file. It provides a message interface for sending `mblks` to the sound driver process as well as receiving them. Additionally it provides an API, which primarily creates messages which are then sent to the sound driver process.

9.1 Overview

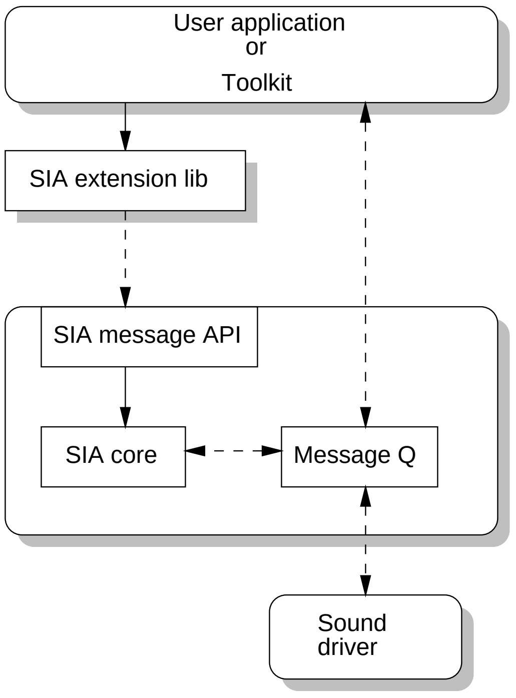


Figure 21: SIA overview

Figure 21 shows an overview of the integration of SIA between the user application and the sound device driver. The SIA core controls the flow of messages in the SIA message queue which makes it possible to manipulate data flowing in both directions. The SIA core can also send control messages down to the sound driver process.

9.2 SIA API

SIA provides an API for controlling the audio device. The SIA core generates control messages which are sent down to the sound device driver process. The handling of these messages is completely hidden within the SIA process; the user application is not even aware of messages being sent when using the API. These messages are created using the:

```
int sia_send_cmd_play();
int sia_send_cmd_stop();
int sia_send_cmd_pause();
int sia_send_cmd_resume();
```

functions and result in control messages defined as:

- `ROME_M_SIA_CMD_SETUP`
- `ROME_M_SIA_CMD_PLAY`
- `ROME_M_SIA_CMD_STOP`
- `ROME_M_SIA_CMD_PAUSE`
- `ROME_M_SIA_CMD_RESUME`

to be sent to the sound driver. The sound device driver implements a handler for these messages.

The *setup* command message is sent automatically when the play command is issued or the format of the audio stream changes. It does not need to be created by the user application. Using the SIA API only makes sense, if there are sound samples being played by the audio device. Otherwise the use of the SIA API has no effect and the sent messages will be ignored in the sound driver process.

9.3 SIA messages

An application uses standard ROME `mblks` for sending sound samples to SIA. For each sample the application must send a *chain* of two `mblks` connected via the `b_cont` field. The first `mblk` contains the a defined *header structure* which is recognized by the SIA core. The header structure contains information about the sound sample such as *sample frequency* or *sample bit width* and must be provided for SIA to work. The second `mblk` contains the actual sound sample data. The application can request a buffer from SIA or provide its own buffer and just set up the `mblk`'s *read* and *write* pointers to its local buffer.

This mechanism is more efficient as the user application can *reuse* the header `mblk` and only needs to update the `b_cont` field when sending a new `mblk` to SIA.

9.4 Generic sound sample structure

SIA abstracts the sound device by introducing a generic sound sample structure. It is defined as a C-structure which is located in the *external* SIA header file. SIA expects messages to contain a header of the specified format. This generic structure defines the possible attributes of a *virtual audio device* (see section 9.6 below).

9.5 Multiplexing

The SIA adapter process can be used by many user applications in parallel. It currently only implements one message queue for all messages which means that messages from applications are handled on a *first come first served* policy. Downstream messages are checked for their content before being forwarded to the device driver process and *control messages* are inserted into the message stream, if playback formats change.

This policy is adequate for current applications running on ROME, but future systems may require a more sophisticated approach.

9.6 Virtual audio device

All audio data passes through the SIA process. A level of abstraction in SIA defines a *virtual audio device* which is presented to the upper layers by the SIA (much like the virtual devices in GIA). This audio device has generic properties and can be used by applications to play any sound sample that meets the virtual device requirements, such as sample frequency or sample bit width.

The current virtual device supports only 8 bit, mono, 22040 hertz sound samples.

9.6.1 Audio stream transformation

A next step could be to transform the audio stream on the fly. The virtual device would be less restrictive, allowing any type of audio data to be sent through SIA. The SIA core would transform the audio data before forwarding it downstream, adapting its parameters to the capabilities of the physical audio device. This may be applicable for a multi-way conferencing application where participants may interrupt and multiple “voices” must be presented at once.

9.6.2 Audio stream merging

Usually only one sound sample can be played by a single audio device. However, SIA could implement a mechanism that allows multiple user processes to open a device and playback their sound samples. SIA could merge sound samples from different processes into one audio stream, also using audio stream transformation, and send the stream to the audio device driver.

10 RWM - ROME Window Manager

When running multiple applications with multiple windows on one system, it is necessary to provide the user with the ability to manipulate windows on the screen. The task of GIA is to manage drawable areas on the screen. It does not provide any *window related* functionality. GIA created drawables and displays them on the screen as they are created, using default screen coordinates and stacking order.

The ROME window manager extends the functionality of GIA by presenting well defined shared library interface to GIA. It implements the functionality to decorate windows, move them around on the screen, resize them or change the stacking order of windows. GIA uses the shared library interface at *strategic* points in the code.

10.1 RWM overview

Unlike window managers that can be found on UNIX or UNIX like systems the ROME window manager is not a process. It is implemented as a shared library as an extension to GIA. The reason for using a shared library instead of a process are:

- The memory footprint is very small
- Communication via an API is faster than inter process communication via messages
- Using a shared library requires less administrative effort than using a process
- The advantages of flexibility that would be provided by implementing it as a process are not required for ROME

10.2 Window focus

When working in an environment with multiple windows it is necessary to define an *active* window that currently receives inputs from the user. If a window is *active* the window manager gives it the *focus*. If a window gets focused, RWM automatically puts it in top of the desktop and changes its color to the “focused color”. As it is put on top of the stacking order it will be repainted and be fully visible.

A window can receive the focus in many ways. If it is the only window on the desktop, it obviously must have the focus. When multiple windows are on the desktop, the user can change the focused window by selecting it. Newly created windows will always be focused once they appear on the screen.

10.3 RWM–GIA interaction

Whenever GIA processes a mouse, keyboard or drawing event it will send the information to RWM before processing it itself. This way RWM always knows about user inputs and drawing requests. RWM might change the information according to its needs or keep it untouched and just recognize it.

As RWM is a shared library GIA can not communicate with it using messages. RWM provides a well defined API that allows GIA to "inform" RWM about events like mouse clicks or window events. GIA heavily interacts with RWM and also relies on RWM feedback. Table 1 gives a short overview how GIA and RWM interact.

Event in GIA	RWM API function used	RWM action
New drawable created	`rwm_size_hint()`	Calculates and adjusts the size of thedrawable so RWM is able to paint itswindow decorations into the drawable. Sets the initial position of the newwindow.
	`rwm_window_new()`	Creates a window structure inside RWM and links it in to the RWM core list.This function is only called by GIA,if the drawable is a *main window*drawable.
Drawable deleted	`rwm_delete_window()`	Removes the window structures from the RWM core list.
Drawable size changed	`rwm_resize_window()`	Calculates and adjusts the new size of thedrawable and returns the result.
Drawable mapped	`rwm_map_window()`	Raises the drawable, paints its decorationsand gives the window the focus.
Mouse button pressed	`rwm_mouse_button_pressed()`	Checks, if any window is affectedby the mouse click and returns`RWM_HANDLED` if so.
Mouse button released	`rwm_mouse_button_released`()	Checks if any window isaffected by the mouse button releaseand returns `RWM_HANDLED` if so.
Key pressed	`rwm_keyboard_event`()	Checks, if any window isaffected by the keystroke andreturns `RWM_HANDLED` if so.

Table 1: GIA–RWM interaction

To create a system without RWM a "NULL-RWM" can be used. It provides enough functionality for GIA to work properly, implementing a defined *default behaviour* and return values without providing a typical window manager functionality.

10.3.1 Creating a window

When GIA creates a drawable, it requests a *size hint* from RWM. RWM now has the possibility to change the size of the drawable, so there is enough space for its window decorations. GIA passes a structure to RWM which contains the *graphic context* of the window *client area* and the window *frame*. RWM basically enlarges the frame context and offsets the client area context.

GIA is not aware of changes that RWM made in the attributes structure, as it only uses the client area graphics context for drawing.

RWM creates a window structure which contains a reference to the drawable that is associated with that window. On future requests this information can be looked up in the RWM internal window list.

10.3.2 Deleting a window

Deleting a window is straightforward. If RWM is notified that a main window drawable has been deleted it simply deletes the associated window structure from its list.

10.3.3 Painting window decorations

When a drawable is mapped onto the screen, GIA calls the RWM *map* function. RWM will draw the decorations for the window into the (offscreen) drawable buffer, raise it to the top of the stacking order and assign it the focus. When finished it will return control to GIA which will paint the window on the screen.

10.3.4 Processing mouse actions

Mouse actions are also forwarded to RWM. RWM looks through the visible windows in its list and checks if a mouse operation occurred in an *active area* like the titlebar of a corner of the window. If an action affects a window, RWM takes care of giving user feedback by for example painting a *ghost window* or other graphical elements.

10.3.5 Processing keyboard actions

RWM processes keystrokes by checking if there is any function associated with it for the currently focused window. If so, the function will be applied.

10.4 RWM configuration

It is possible to configure RWM windows to some extend. Configuration is done through the system color configuration library which contains the default colors that are used for active or inactive window decorations or titlebar font color. The color library values are predefined at compile time but can be changed during run-time by using the color database library API.

11 The Device Driver Layer

Most of the components in RMP are designed to be hardware independent by using the concept of virtual devices and device abstraction. The basic hardware abstraction is done in the adapter layer, channeling all accesses to the hardware through adapter processes.

The device driver layer contains the hardware dependent components of RMP, the *device driver processes*. Device driver processes directly access the hardware of the system at a *register level*, exploiting the full capabilities of the available chipsets. They communicate with adapter layer processes via ROME messages and events.

11.1 Device driver initialization

Most devices must be initialized before they can be used. This can usually be done at boot time, except for dynamically added devices like PC Cards. As device drivers are processes, the ROME startup code will call their *init* function when the system is started. This gives the device driver the opportunity to initialize both, *hardware registers and* software *data structures* which need to be set to a defined state. It can, for example, set up interrupt handlers and enable the interrupt lines for a device.

The ROME startup code calls the device's *init* functions one after another in the order in which they are entered into the system *process table*. If there are dependencies between devices' setups it is the programmers responsibility to ensure a correct order when building a system.

11.2 Device classes

As described in section 7.2, RMP groups devices into device classes to define virtual devices with generic properties. These device classes describe a set of devices with similar features such as *pointing devices*, *screens* or *character input devices*. Devices of the same class will behave the same way, independent of their hardware structure. This makes it possible to exchange a hardware component and the associated hardware device driver without changing other code.

For example, the graphics driver for the WebPanel LCD screen uses the same communication interface as the driver of the Number9 video card, although the hardware is completely different.

11.3 Communication interface

Device driver processes communicate with processes in the adapter layer by ROME messages or events. The message and event types are defined by the adapter process *not* the device driver process. A device driver must support the set of messages and events that are defined by the adapter process in order to provide a generic interface for the adapter process. The types and structures of the messages and events are defined on a per-device class basis. They are defined in the *Build* file of the adapter ROME module, for example:

```
Event GIA_MOUSE
{
    uint event_type;
    void *drawable;

    int x;
    int y;
}
```

defines an event for the mouse *pointing device*.

Additionally, device driver processes must provide a certain set of *support* messages for the adapter layer to be able to determine hardware capabilities. These messages are also defined per device class.

12 Graphics device driver

Graphics device drivers in RMP are relatively simple processes. As most of the more complex functionality is implemented in the hardware independent GIA adapter layer process, the graphics device driver only needs to implement elementary functionality such as:

- copy an image from a buffer *to* video memory
- copy an image *from* video memory to a buffer

This makes the development of graphics device drivers very easy.

12.1 Message types

As GIA abstracts screen devices it also provides the definition of the message types and structure that must be supported by a graphics device driver process. The basic messages are designed to send image buffers to the device driver process. They contain the image buffer, the size of the image and the coordinates of where the image should be displayed on the screen. To increase performance, the image buffer is *not copied* into the message, but the message contains a *pointer reference* to the image buffer. So there will be *only one* copy operation when the image buffer is copied into video memory.

There also exists a support message to retrieve the device capabilities. Currently GIA defines the following messages:

`ROME_M_GI_PUT` Copy the image rectangle into video memory at the defined coordinates;

`ROME_M_GI_PUT_MASK` Same functionality as `ROME_M_GI_PUT`, but treat the color *0 (zero)* as a *transparent* color;

`ROME_M_GI_XOR` Logical XOR the image rectangle into the screen;

`ROME_M_GI_OR` Logical OR the image rectangle into the screen;

`ROME_M_GI_AND` Logical AND the image rectangle into the screen;

`ROME_T_GI_GET` Retrieve the defined image rectangle from the screen;

`ROME_T_GI_CAPS` Retrieve the device capabilities such as screen *x* size, screen *y* size and *color depth.*

The message structures are defined in the GIA *Build* file and will be translated into ROME messages which are located in the file:

```
Messages.h
```

in the global include directory.

12.2 Message handling

To increase performance, ROME processes can handle messages in their queue handlers. However, as queue handlers are executed within a *critical section* all interrupts are disabled when running in the queue handler, which means that the system can not respond to any interrupts while the processor is running in a process' queue handler. Therefore only messages that require little computation time should be completely handled in the process' queue handler.

A graphics device driver usually handles large blocks of image data. Assuming a 640x480 image with 16 bit color depth, the device driver has to copy 600 kibibyte into video memory and might even have to apply a logical function on the data requiring that the current data be accessed in a read-modify-write cycle. Even on a high performance system this needs some time.

To prevent the system from being blocked during this time, the graphics device driver *must not* handle dataflow messages in its queue handler. All such messages *must* be handled in the device driver main process message loop.

12.3 Graphic device capabilities

GIA needs to know the graphics device capabilities to be able to provide it with the appropriate image buffer format. When GIA is configure for a device it requests the capabilities from the graphics device driver by sending a `ROME_M_GI_CAPS` message. Therefore all graphics drivers must support this message. The structure of the message is defined as:

```
typedef struct
{
    int dx;
    int dy;
    int depth;
    int format;
}
ROME_T_GI_CAPS;
```

The fields of the structure must be set by the device driver according to the graphic device capabilities:

dx screen resolution x (in pixels)

dy screen resolution y (in pixels)

depth color depth (bits per pixel)

format color format (e.g. RGB 565)

12.4 Color depth

The low level graphics driver relies on the image data passed in the message. It does not convert any image data into the appropriate color format. GIA is responsible for setting up the image buffer for the right size, format and color depth (see section 8.3.3). The only operation that is applied to the image data is the operation defined by the message opcode.

As color conversion happens in the upper layer, the driver code remains small and relatively simple making the development of new drivers or porting to other hardware platforms easier.

13 Pointing device driver

In a graphical environment with multiple windows on a single screen, the user needs to interact with the system and manipulate graphical elements on the screen. An easy way to manage this is to provide a *pointing device* which itself is represented on the screen as a small graphical element, called *mouse cursor*. The *device hardware* allows it to move the mouse cursor around on the screen. By doing so the user can select items and manipulate or interact with them.

In building a ROME system there are many possibilities for attaching a *pointing device*. The most common one might be the use of a conventional mouse, but touchscreens and pen input devices are getting more common, especially in embedded, mobile systems.

13.1 Event type

The driver recognizes movement of the attached pointing device through its interrupt handler. The interrupt handler also handles changes of buttons, switches or other hardware attached or related to the pointing device. It encapsulates changes in the devices' state into a pointing device *event*. The event is defined by the mouse adapter process, as it implements the first abstraction level of the pointing device. The event is defined as:

```
ROME_E_MOUSE
```

and encapsulates the structure:

```
typedef struct
{
    int event_type;

    int xvalue;
    int yvalue;
    int touch;
}
ROME_P_MOUSE;
```

13.2 Device classes

RMP groups pointing devices into two different classes:

- Devices sending *absolute* coordinates
- Devices sending coordinate *deltas*

Device driver processes of either class are communicating with the mouse adapter process. Depending on its configuration, the mouse adapter process translates the coordinate values provided by the pointing device driver into a generic format.

However, devices sending absolute coordinates have to provide *real* screen coordinates. This means, they have to transform their internal physical coordinates into screen coordinates. To do so they need to be configured for the screen size and screen orientation.

13.3 Device configuration

Depending on the device type there can be different requirements for configuring the device. Some devices do not need any configuration at all, other devices need extensive configuration involving user interaction. For example, a serial mouse using coordinate deltas to report its movement would not necessarily have to be configured at all, whereas a touch panel might need to be set up by defining reference coordinates and require user feedback.

13.4 Configuration processes

Some pointing devices require more complex configuration procedures. For example, for the configuration of a joystick, *targets* have to be shown on the screen while awaiting a user feedback. This functionality is implemented in a separate process, a configuration process. Usually this process is run at startup time and terminates after the device is configured. As it is a normal ROME process it can take advantage of the full RMP functionality.

The *Gunze* device driver for the WebPanel touchscreen uses such a configuration process. The configuration process requests the user to press two *targets* and awaits the *raw* coordinates from the driver. Then it sends a configuration command to the *Gunze* driver, providing it with configuration parameters. After that the *Gunze* device driver is able to translate its physical touchscreen coordinates into *real* screen coordinates and send them to the mouse adapter process.

The *Gunze* configuration process itself is configured through the system manager configuration file. The entry in the configuration file defines the mouse adapter process name and the selected screen orientation, for example:

```
[boot]:
   :mconfig:mouse 1
```

The system manager will send a command to the configuration process, initiating the configuration procedure.

14 Support libraries

RMP currently includes three support libraries. One is used by STK, the other two are used by RTK and GIA. The purpose of these libraries is to provide functionality while keeping the modular RMP design and making it easy to add new functionality.

14.1 XPM

X PixMap (XPM) is an image format commonly used in X11 systems. As almost every graphic application supports XPM format, it is very useful to implement an XPM library for ROME. The XPM library supports the basic functionality for creating images from files and compiled in XPM structures.

14.1.1 Format

XPM images are defined in form of a C-structure. It is possible to include a XPM image file into a C-file the same way as including a header file. The XPM file defines a *static* C variable which can be accessed within the C-module in which it is included.

The variable defines an array of *zero terminated strings* which describe the image. These strings contain information about the image size, color depth, color table and the actual bitmap. An application can easily scan these strings and create an image buffer.

The XPM library provides functions to scan loaded and included XPM files. These functions create an image structure which is used by the application to handle XPM images. The structure is defined as:

```
typedef struct
{
    int dx;
    int dy;
    int depth;
```

```
        int mode;
        char *image;
    }
    image_t;
```

The XPM functions *do not* allocate memory for the image structure. Applications must provide structures which are used by the XPM functions. It is possible to set one of two different *modes* of the created image:

XPM_MODE_NORMAL The image data is used ***as is.*** All colors will be displayed as they are defined in the XPM file.

XPM_MODE_MASK The color ***zero (0)*** is interpreted as a *transparent* color. This makes it possible to define *shaped* images.

14.1.2 Included XPMs

An application can include XPM files directly into the C-modules. This makes sense if:

- The system does not support a filing system and it is therefore not possible to read XPM files from disk;
- The XPM image files are reasonable small.

However, even if an XPM file is included into the C-module, it can not be used directly. Before it can be handled it has to be transformed into an XPM image format. To do this XPM library provides the:

```
    int xpm_image_from_data(image_t *img, char **xpm);
```

function.

14.1.3 File support

Loading XPM files from disk makes sense when the images are very big. As XPM images are stored in *string* format including them into C-code results in huge data sections in the target files. Therefore it is useful to be able to load images from disk.

The XPM library provides the:

```
    int xpm_image_from_file(image_t *img, char *filename);
```

function to create an image buffer from a XPM file stored on disk. It the image buffer is only needed temporarily, for example, to display an icon in a dialog, it can be deleted afterwards. This saves resources in memory critical applications.

14.1.4 Memory images

Some applications might want to compute images without using XPM files. For example, a video application could receive a video stream and wants to display it on the screen. The XPM library provides a function to create an image from a memory buffer:

```
    int xpm_image_from_mem(image_t *img, int dx, int dy, int mode, char *buf);
```

The application must provide all properties of the image, such as size, color depth, mode and the image buffer.

14.2 Audio library

The audio library is used by STK and currently simplifies handling of sound files on a filing system. It abstracts the sound files to a handle on which the application can apply actions. The handle is defined as:

```
    H_AUDIO
```

The handle must be provided on any operation to identify the audio file.

14.2.1 Basic functions

The audio library provides basic functions to work with audio files. It is possible to *open* and *close* audio files using the:

```
H_AUDIO audio_open(const char *filename, const char *mode);
void audio_close(H_AUDIO haudio);
```

functions.

To read from the audio file the functions:

```
size_t audio_read(H_AUDIO haudio, char *buf, uint len);
int audio_reset_fpos(H_AUDIO haudio);
```

can be used.

14.2.2 Info structure

Applications can request information about an opened audio file. The audio file properties are returned in an audio info structure which is defined as:

```
typedef struct audio_info_d
{
    ushort format;    /* sample format */
    ushort channels;  /* number of channels */
    uint rate;        /* sample rate */
    ushort bits;      /* bits per sample */
    uint datasize;    /* size of the data (bytes) */
}
audio_info_t;
```

It can be requested using the:

```
const audio_info_t *audio_info(H_AUDIO haudio);
```

function.

14.3 Color database

To keep a consistent look and feel between different applications it is necessary to define a common color scheme. The color database library provides an API to define colors for different elements on the screen such as buttons or window frames. Applications can use these entries to set the color of their graphical elements. RTK, for example, uses the color database library to set the color of all its widgets. Therefore, any application which uses RTK will have the same look.

14.3.1 API

The color database API is quite simple. It provides two functions:

```
uint color(color_t idx);
void color_set(color_t idx, uint col);
```

To *request* and *set* a color. The application must provide the *index* of the requested color. Changes in the color database *do not* automatically cause the current screen to be updated. An application will have to update its windows in order to make the color changes visible.

14.3.2 Color index

The color database provides a set of index definitions for different graphical elements on the screen. These definitions are located in the file:

```
colors.h
```

It is possible to add new definitions for future elements. The default values of the database color entries are hard-coded into the library (file `colors.c`).

14.3.3 Color translation

The color database also provides macros to do color translation. Currently only a macro for translating 24 bit to 16 bit RGB 565 is provided:

```
COLOR_24TO16(_c)
```

These macros are mainly used by GIA for the WebPanel LCD display.