

# Création de signaux pour vos widgets GTK+

par [Franck.H](#)

Date de publication : 27/09/2006

Dernière mise à jour :

Création d'un signal pour un widget personnel !

- I - Introduction
- II - Pré-requis
- III - Les signaux
  - III-A - Un bref rappel
  - III-B - Plan de route de notre signal
- IV - Modification du code source
  - IV-A - Le fichier gtkled.h
  - IV-B - Le fichier gtkled.c
  - IV-C - Signature de notre callback
- V - Conclusion
- VI - Code source complet
- VII - Remerciements

## I - Introduction

Dans le cours précédent [Créer son widget GTK+ en Langage C](#) (qu'il convient de lire avant de suivre ce tutoriel), nous avons créé un widget à partir de rien ; en fait plus ou moins car on part tout de même d'un GtkWidget.

Un widget personnel c'est bien, dans notre cas, GtkLed nous permet d'afficher Deux états: Eteint et Allumé ... Tout ceci est bien me direz-vous mais il faut passer par un GtkButton pour pouvoir lui changer l'état courant.

Dans ce tutoriel je vais vous expliquer comment créer un signal personnel qui nous permettra de changer l'état de notre GtkLed par un simple clic sur ce widget !

Si vous avez des questions concernant GTK+ l'équipe de [developpez.com](#) sera ravie de vous aider, rendez-vous sur le [Forum GTK+ de developpez.com](#)

## II - Pré-requis

- Avoir suivi le cours [Créer son widget GTK+ en Langage C](#)
- Posséder l'archive [GtkLed.zip](#)

## III - Les signaux

### III-A - Un bref rappel

Si vous savez ce que sont exactement les signaux GTK+ vous pouvez sauter ce chapitre et pour les autres...

Les signaux GTK+ sont plus ou moins identiques aux messages d'événements de l'API Win32 ; Si vous avez un peu étudié cette API, vous savez que ce sont des constantes symboliques représentant des entiers (non signés, longs ... c'est suivant l'API en fait). Ces constantes permettent de gérer un événement utilisateur (clic sur un bouton par exemple) ou des événements du système lui même (dessin du contenu d'une fenêtre).

Avec GTK+, nous nous référons aux signaux par le biais d'une chaîne constante mais les signaux eux-mêmes n'en restent pas moins que des entiers ; pour être plus exacte, ce sont des **quint** (*unsigned int, entiers non signés*).

Il faut savoir que les signaux GTK+ sont internes à cette bibliothèque mais sont reliés directement avec les signaux du système sous-jacent ; les signaux internes au noyau d'un système passent par des tubes (*pipes*) comme c'est le cas des signaux du serveur X11 (serveur graphique de Linux).

La différence essentielle avec le traitement des événements avec GTK+ (par rapport à l'API Win32 par exemple), est que nous pouvons lier chaque signal à une fonction unique ce qui offre l'avantage de découper le comportement de l'application, cela rend également le code plus lisible alors que sous l'API Win32 nous n'avons qu'une unique fonction qui récupère tous les messages (utilisateurs et système).

Il nous suffit d'utiliser la fonction:

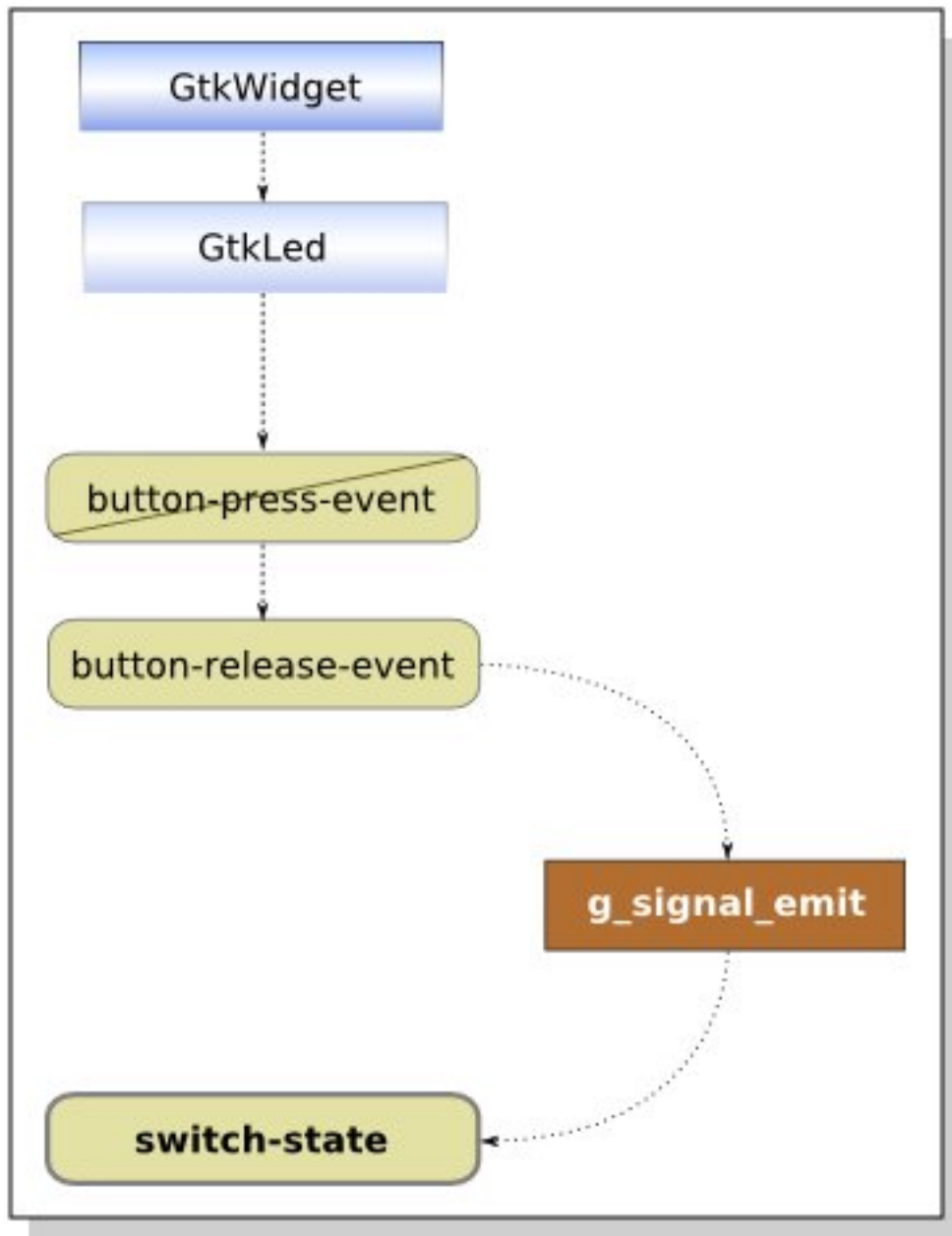
```
gulong g_signal_connect(gpointer *object, const gchar *name, GCallback func, gpointer func_data );
```

pour lier une fonction à un signal GTK+. Il est à noter également que chaque signal GTK+ possède un comportement par défaut qui la plupart du temps est de rien faire, sauf pour certains événements comme par exemple le signal **expose-event** qui permet de faire redessiner le widget lorsque l'API le lui demande mais, nous pouvons bien sûr redéfinir son comportement nous même en connectant ce signal avec une fonction de rappel (*ou callback*) personnelle !

Nous ne faisons ici que gratter la surface et nous voyons déjà la puissance et la souplesses de la gestion des signaux de GTK+ !

### III-B - Plan de route de notre signal

Comment va fonctionner notre signal ? En réalité, nous allons le faire appeler par un autre événement, voyons d'abord un petit schéma qui nous montre son cheminement:



Notre widget `GtkLed` héritant de `GtkWidget`, nous avons une pléthore de signaux à notre disposition ; nous, nous allons faire reconnaître les signaux **`button-press-event`** et **`button-release-event`** par notre widget (par défaut notre widget ne gère pas les événements utilisateurs). Sur le schéma vous pouvez voir que l'événement **`button-press-event`** est barré ! En effet, nous devons le faire intercepter par notre widget mais nous n'allons pas le gérer, nous laisserons donc ce signal à son comportement par défaut: Rien faire !

Donc comment cela va-t-il se passer ... Nous allons connecter notre widget `GtkLed` au signal **`button-release-event`** qu'il hérite de `GtkWidget` ; ce signal étant directement relié à une fonction privée **`gtk_led_button_release`**, que nous verrons plus loin dans ce cours, nous allons simplement faire déclencher notre signal **`switch-state`** par un appel de la fonction **`g_signal_emit`** dans notre callback privé (*`gtk_led_button_release`*).

Cette façon de procéder va vous faire croire que c'est notre signal personnel qui va nous permettre de directement cliquer sur le widget alors qu'en fait c'est un autre signal qui se trouve dans GtkWidget qui va gérer le tout mais de façon transparente !

Vous me direz sans doute: Quel intérêt de faire un signal de ce genre car nous disposons déjà de fonctions pour changer l'état du widget ? En fait, c'est d'une part déjà plus intuitif et d'un autre côté, cela nous permet de faire d'autres actions relatives au changement d'état du widget sans devoir également gérer le changement de ces états !

## IV - Modification du code source

### IV-A - Le fichier gtkled.h

Ouvrez le fichier **gtkled.h** et rendez-vous dans la structure **\_GtkLedClass**. Ici, il nous suffit d'ajouter un pointeur de fonction pour notre signal. Ci-dessous notre structure complète:

```
struct _GtkLedClass {
    GtkWidgetClass parent_class;

    /* Signal personnel. */
    void (* switch_state) (GtkLed * led);
};
```

Voilà, c'est tout pour ce fichier ! La signature des signaux sont toujours sous cette forme, une fonction qui renvoie (ou ne renvoie pas) de valeur et qui prend comme paramètre un pointeur sur le type du widget.

### IV-B - Le fichier gtkled.c

Nous allons maintenant créer notre signal, nous le nommerons **switch-state** tout simplement, je pense que c'est un nom assez explicite donc pas la peine de donner des explications !

Commençons par créer la signature de notre fonction qui va gérer en interne le changement d'état de notre widget. Au début de la liste des fonctions statiques, ajoutez notre nouvelle fonction:

```
static gboolean gtk_led_button_release (GtkWidget * widget,
                                         GdkEventButton * event);
```

Pour notre signal, il nous faut un tableau d'entiers non signés, soit, créons-le ; à la fin de la liste des prototypes des fonctions privées, ajoutez donc ceci:

```
enum {
    GTKLED_SWITCH_STATE_SIGNAL,
    GTKLED_NB_SIGNALS
};

static guint gtkled_signals [GTKLED_NB_SIGNALS] = { 0 };
```

J'ai ajouté une énumération de constantes pour une question de compréhension du code source, habitude à prendre surtout si vous créez plus d'un signal. Nous déclarons ensuite simplement un tableau de **guint** (*unsigned int*) puis nous initialisons tous ses indices à la valeur zéro. Ce tableau servira en réalité à stocker l'identifiant unique de notre signal une fois créé.

Nous allons maintenant implémenter notre fonction **gtk\_led\_button\_release** juste en-dessous de la fonction **gtk\_led\_new** comme suit:

```
static gboolean
gtk_led_button_release (GtkWidget * widget,
                        GdkEventButton * event)
{
    GTK_LED (widget)->state = !GTK_LED (widget)->state;
    gtk_led_paint (GTK_WIDGET (widget));
}
```

```

g_signal_emit (
  G_OBJECT (widget),
  gtkled_signals [GTKLED_SWITCH_STATE_SIGNAL],
  0
);

return FALSE;
}

```

Rien de bien compliqué ici ! Nous inversons l'état courant de notre widget puis nous le faisons se redessiner en appelant la fonction **gtk\_led\_paint**.

La nouveauté réside dans l'appel à la fonction **g\_signal\_emit**. En fait, nous demandons à ce que notre signal **switch-state** soit déclenché à partir du signal **button-release-event** ce qui nous permettra de faire changer l'état du widget en cliquant simplement dessus !

Nous entrons maintenant dans le vif du sujet en créant notre signal personnel. La création de notre signal va se faire dans la fonction **gtk\_led\_class\_init** comme suit à la fin de la fonction:

```

gtkled_signals [GTKLED_SWITCH_STATE_SIGNAL] = g_signal_new (
  "switch-state",
  G_TYPE_FROM_CLASS (klass),
  G_SIGNAL_RUN_FIRST | G_SIGNAL_ACTION,
  G_STRUCT_OFFSET (GtkLedClass, switch_state),
  NULL, NULL,
  g_cclosure_marshal_VOID__VOID,
  G_TYPE_NONE,
  0
);

```

Cette fonction peut sembler compliqué au premier abord mais il n'en est rien. Voici son prototype:

```

guint g_signal_new (const gchar *signal_name,
                  GType itype,
                  GSignalFlags signal_flags,
                  guint class_offset,
                  GSignalAccumulator accumulator,
                  gpointer accu_data,
                  GSignalCMarshaller c_marshallier,
                  GType return_type,
                  guint n_params,
                  ...);

```

- Le premier argument n'est autre que le nom du signal tel que nous l'appellerons avec la fonction **g\_signal\_connect**.
- Le second argument est le numéro unique du widget qui lui est attribué lors de son enregistrement auprès de GTK+.
- L'argument **signal\_flags** sert à déterminer si le gestionnaire par défaut doit être lancé avant ou après celui de l'utilisateur. Il faut au minimum indiquer le drapeau **G\_SIGNAL\_RUN\_FIRST** ou **G\_SIGNAL\_RUN\_LAST** qui sont les constantes généralement utilisées. [Les autres constantes possible](#)

Cet argument sert à déterminer en partie, le moment où sera appelé le gestionnaire de l'utilisateur (notre fonction dans la structure `_GtkLedClass`) que nous attachons à notre widget avec l'argument suivant (`class_offset`) !

- L'argument **guint class\_offset** n'est autre que l'adresse du pointeur de fonction pour notre signal dans la structure de classe de notre widget. La fonction utilisée: **G\_STRUCT\_OFFSET** prend pour argument le nom de la structure de classe puis le nom du pointeur de fonction de notre signal.
- L'argument **GSignalAccumulator accumulator** est en fait une fonction callback spéciale qui permet de collecter des valeurs de retour d'autres callback du même widget pendant une émission de signaux, nous laissons cette argument à **NULL**.
- L'argument qui suit n'est autre qu'un pointeur sur des données utilisateur tout comme les autres fonctions de rappel dans GTK+. Vu que nous n'utilisons pas de fonction d'accumulation nous laissons cet argument à **NULL**.
- L'argument **GSignalCMarshaller c\_marshall** permet de spécifier la signature de la fonction interne à GTK+ qui s'occupe de lancer le signal lorsque nous appelons **g\_signal\_emit**. Dans notre cas, **g\_cclosure\_marshal\_VOID\_\_VOID** dit à GTK+ que nous utilisons une fonction qui ne prend pas d'argument supplémentaires que ceux par défaut (nous verrons la signature de notre fonction de rappel plus loin) et ne renvoie pas de résultat.
- L'argument **GType return\_type** est simplement le type de retour de notre signal, ici **void** donc nous utilisons la constante **G\_TYPE\_NONE**. Il existe d'autres constantes utilisables, les voici:
  - **G\_TYPE\_INVALID**
  - **G\_TYPE\_NONE**
  - **G\_TYPE\_INTERFACE**
  - **G\_TYPE\_CHAR**
  - **G\_TYPE\_UCHAR**
  - **G\_TYPE\_BOOLEAN**
  - **G\_TYPE\_INT**
  - **G\_TYPE\_UINT**
  - **G\_TYPE\_LONG**
  - **G\_TYPE\_ULONG**
  - **G\_TYPE\_INT64**
  - **G\_TYPE\_UINT64**
  - **G\_TYPE\_ENUM**
  - **G\_TYPE\_FLAGS**
  - **G\_TYPE\_FLOAT**
  - **G\_TYPE\_DOUBLE**
  - **G\_TYPE\_STRING**
  - **G\_TYPE\_POINTER**
  - **G\_TYPE\_BOXED**
  - **G\_TYPE\_PARAM**
  - **G\_TYPE\_OBJECT**
- Le dernier argument permet de spécifier la liste des arguments supplémentaires à rajouter à notre signal, ici nous n'avons pas besoin de signaux en plus, ceux par défaut qui sont définis explicitement par GTK+ à savoir, le widget qui récupère le signal et les données utilisateurs, nous suffisent.

Pour nous permettre de pouvoir cliquer sur notre widget, nous devons le lier à un signal, c'est ce que nous faisons dans la fonction **gtk\_led\_init**, à la fin de la fonction:

```
static void
gtk_led_init (GtkLed * led)
{
    led->on_image = NULL;
    led->off_image = NULL;
    led->on_pixbuf = NULL;
    led->off_pixbuf = NULL;
    led->width = 0;
    led->height = 0;
    led->state = FALSE;

    g_signal_connect (
        G_OBJECT (led),
        "button-release-event",
        G_CALLBACK (gtk_led_button_release),
        led
    );
}
```

Pour finir, nous devons donner la possibilité à notre widget d'intercepter le signal **button-press-event** et **button-release-event** ; rendez-vous pour cela dans la fonction **gtk\_led\_realize**, retrouvez la propriété **attributes.event\_mask** puis changez-la par celle-ci:

```
attributes.event_mask = gtk_widget_get_events (widget) |
    GDK_EXPOSURE_MASK | GDK_BUTTON_RELEASE_MASK | GDK_BUTTON_PRESS_MASK;
```

## IV-C - Signature de notre callback

Nous y arrivons, vous avez enfin le droit de connaître la signature de votre premier signal. Ici rien de nouveau, nous reprenons simplement la signature de la plupart des signaux:

```
void user_function (GtkWidget * widget, gpointer data);
```

Comme à l'accoutumée, vous remplacez juste **user\_function** par un nom plus explicite lorsque vous utiliserez ce signal ... comme d'habitude vous me direz ;)

## V - Conclusion

Et voilà, nous en avons fini avec ce tutoriel, j'espère qu'il vous aura bien servi et éclairci les choses !

Comme vous avez pu le voir, il n'y a vraiment rien d'insurmontable mais certaines choses sont effectivement à savoir pour pouvoir créer vos propres signaux.

## VI - Code source complet

Ici, vous pouvez télécharger le code source complet du widget livré avec un programme d'exemple, ainsi que le Makefile Linux et un projet Code::Blocks version Windows: [GtkLed-2.zip](#)

## VII - Remerciements

Merci à gege2061 pour ses conseils et à fearyourself pour sa relecture et correction !