

---

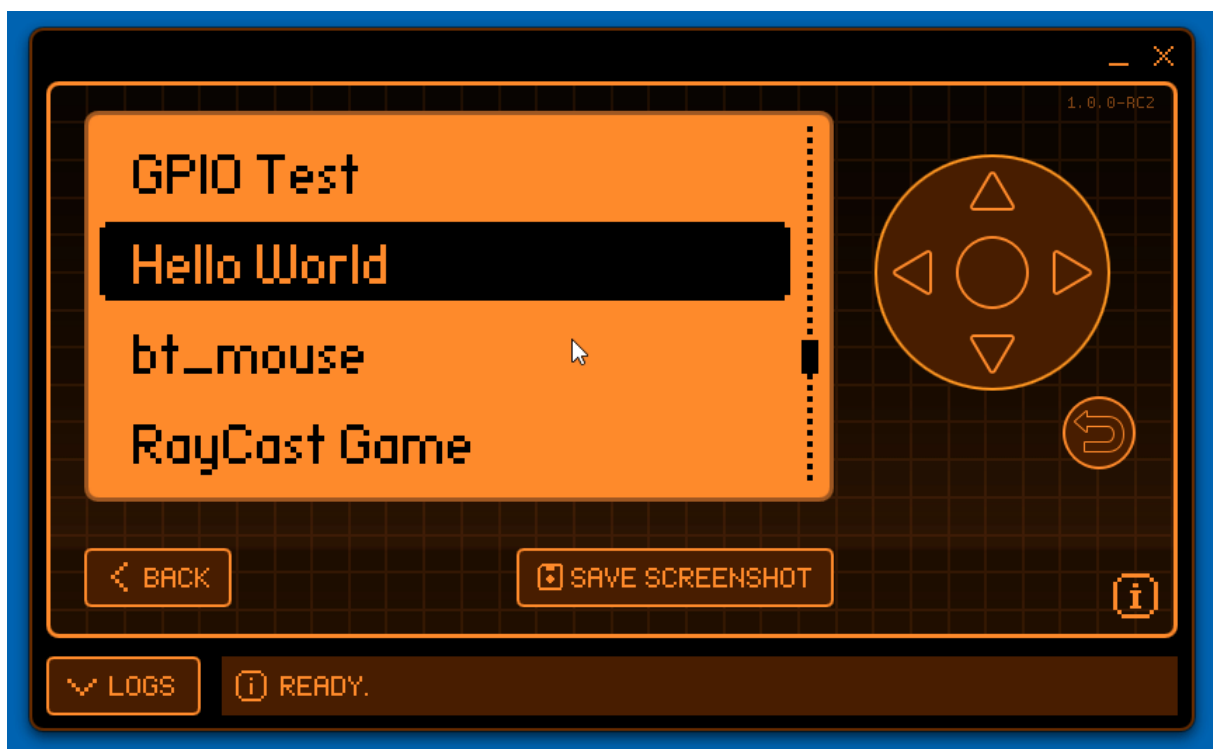
## Flipper-Plugin

Tutorial on how to build a basic “Hello world” plugin for Flipper Zero.

This tutorial includes:

- Sourcecode template for a custom flipper plugin!
- A step by step story on how to create the base of a custom flipper plugin. (There are many other ways of creating apps for flipper, this example is based on the existing snake app)

The tutorial was written during development of flappybird for flipper.



## Hello World - The story

**This is the step-by-step story version of the tutorial. You can skip this and directly continue to the sourcecode if you know what your doing. Make sure you don't forget to add the application to the makefile, and register its functions in `applications.c` (see chapter: Building the firmware + plugin)**

In this tutorial a simple hello world plugin is added to flipper. The goal is to render something in the screen, and make the buttons move that object. In this case it will be the classic “Hello World” text.

---

## Downloading the firmware

1. Clone or download flipperzero-firmware.

```
1 git clone https://github.com/flipperdevices/flipperzero-firmware
```

2. Create a folder for the custom plugin in `flipperzero-firmware/applications/`. For the hello-world app, this will be: `hello_world`.

```
1 mkdir flipperzero-firmware/applications/hello_world
```

3. Create a new source file in the newly created folder. The file name has to match the name of the folder.

```
1 touch flipperzero-firmware/applications/hello_world/hello_world.c
```

## Plugin Main

For flipper to activate the plugin, a main function for the plugin has to be added. Following the naming convention of existing flipper plugins, this needs to be: `hello_world_app`.

- Create an `int32_t hello_world_app(void* p)` function that will function as the entry of the plugin.

For the plugin to keep track of what actions have been executed, we create a messagequeue. - A by calling `osMessageQueueNew` we create a new `osMessageQueueId_t` that keeps track of events.

The `view_port` is used to control the canvas (display) and userinput from the hardware. In order to use this, a `view_port` has to be allocated, and callbacks to their functions registered. (The callback functions will later be added to the code) - `view_port_alloc()` will allocate a new `view_port`. - `draw` and `input` callbacks originating from the `view_port` can be registered with the functions - `view_port_draw_callback_set` - `view_port_input_callback_set` - Register the `view_port` to the GUI

```
1 int32_t hello_world_app() {
2     FuriMessageQueue* event_queue = furi_message_queue_alloc(8, sizeof(
        PluginEvent));
3
4     // Set system callbacks
5     ViewPort* view_port = view_port_alloc();
6     view_port_draw_callback_set(view_port, render_callback, NULL);
7     view_port_input_callback_set(view_port, input_callback, event_queue
8     );
9 }
```

---

```
9 // Open GUI and register view_port
10 Gui* gui = furi_record_open("gui");
11 gui_add_view_port(gui, view_port, GuiLayerFullscreen);
12
13 ...
14 }
```

## Callbacks

Flipper will let the plugin know once it is ready to deal with a new frame or once a button is pressed by the user.

**input\_callback:** Signals the plugin once a button is pressed. The event is queued in the event\_queue. In the main thread the queue read and handled.

A reference to the queue is passed during the setup of the application.

```
1 typedef enum {
2     EventTypeTick,
3     EventTypeKey,
4 } EventType;
5
6 typedef struct {
7     EventType type;
8     InputEvent input;
9 } PluginEvent;
10
11 static void input_callback(InputEvent* input_event, FuriMessageQueue*
    event_queue) {
12     furi_assert(event_queue);
13
14     PluginEvent event = {.type = EventTypeKey, .input = *input_event};
15     furi_message_queue_put(event_queue, &event, FuriWaitForever);
16 }
```

**render\_callback:** Signals the plugin when flipper is ready to draw a new frame in the canvas. For the hello-world example this will be a simple frame around the outer edges.

```
1 static void render_callback(Canvas* const canvas, void* ctx) {
2     canvas_draw_frame(canvas, 0, 0, 128, 64);
3 }
```

## Main Loop and plugin State

The main loop runs during the lifetime of the plugin. For each loop we try to pop an event from the queue, and handle the queue item such as button input / plugin events.

---

For this example we render a new frame, every time the loop is run. This can be done by calling `view_port_update(view_port);`.

```
1   PluginEvent event;
2   for(bool processing = true; processing;) {
3       FuriStatus event_status = furi_message_queue_put(event_queue, &
4           event, 100);
5
6       if(event_status == FuriStatusOK) {
7           // press events
8           if(event.type == EventTypeKey) {
9               if(event.input.type == InputTypePress) {
10                  switch(event.input.key) {
11                      case InputKeyUp:
12                      case InputKeyDown:
13                      case InputKeyRight:
14                      case InputKeyLeft:
15                      case InputKeyOk:
16                      case InputKeyBack:
17                          // Exit the plugin
18                          processing = false;
19                          break;
20                  }
21              }
22          } else {
23              FURI_LOG_D(TAG, "FuriMessageQueue: event timeout");
24              // event timeout
25          }
26
27          view_port_update(view_port);
28      }
```

## Plugin State

Because of the callback system, the plugin is being manipulated by different threads. To overcome race conditions we have to create a shared object that is safe to use.

1. Allocate a new `PluginState` struct, and initialise it before the main loop.

```
1   typedef struct {
2   } PluginState;
3
4   // in main:
5   PluginState* plugin_state = malloc(sizeof(PluginState));
```

2. Using `ValueMutex` we create a mutex for the plugin state called `state_mutex`.
3. Initialise the mutex for `PluginState` using `init_mutex()`

- 
4. Pass the mutex as argument to `view_port_draw_callback_set()` so we can safely access the shared state from flippers thread.

```
1 typedef struct {
2 } PluginState;
3
4 int32_t hello_world_app() {
5     FuriMessageQueue* event_queue = furi_message_queue_alloc(8, sizeof(
6         PluginEvent));
7
8     PluginState* plugin_state = malloc(sizeof(PluginState));
9     ValueMutex state_mutex;
10    if (!init_mutex(&state_mutex, plugin_state, sizeof(PluginState))) {
11        FURI_LOG_E("Hello_World", "cannot create mutex\r\n");
12        free(plugin_state);
13        return 255;
14    }
15
16    // Set system callbacks
17    ViewPort* view_port = view_port_alloc();
18    view_port_draw_callback_set(view_port, render_callback, &
19        state_mutex);
20    view_port_input_callback_set(view_port, input_callback, event_queue
21    );
22    ...
23 }
```

## Main Loop

Let's deal with the mutex in our main loop. So we can update values from the main loop based on user input. As an example, we will move a hello-world text through the screen. Based on user input.

1. For this we add a `int x` and `int y` to your state.

```
1 typedef struct {
2     int x;
3     int y;
4 } PluginState;
```

2. Initialise the values of the struct using a new `hello_world_state_init()` function.

```
1 static void hello_world_state_init(PluginState* const plugin_state) {
2     plugin_state->x = 50;
3     plugin_state->y = 30;
4 }
```

Call it after allocating the object in the main function.

---

```

1 PluginState* plugin_state = malloc(sizeof(PluginState));
2 hello_world_state_init(plugin_state);
3 ValueMutex state_mutex;
4 ...

```

4. Acquire a blocking mutex after a new event is handled in the queue. Write values to the locked plugin\_state object when user presses buttons. And release when we finish working with the state.

```

1 PluginEvent event;
2 for(bool processing = true; processing;) {
3     FuriStatus event_status = furi_message_queue_put(event_queue, &
4         event, 100);
5     PluginState* plugin_state = (PluginState*)acquire_mutex_block(&
6         state_mutex);
7
8     if(event_status == FuriStatusOK) {
9         // press events
10        if(event.type == EventTypeKey) {
11            if(event.input.type == InputTypePress) {
12                switch(event.input.key) {
13                    case InputKeyUp:
14                        plugin_state->y--;
15                        break;
16                    case InputKeyDown:
17                        plugin_state->y++;
18                        break;
19                    case InputKeyRight:
20                        plugin_state->x++;
21                        break;
22                    case InputKeyLeft:
23                        plugin_state->x--;
24                        break;
25                    case InputKeyOk:
26                    case InputKeyBack:
27                        processing = false;
28                        break;
29                }
30            }
31        } else {
32            FURI_LOG_D(TAG, "FuriMessageQueue: event timeout");
33            // event timeout
34        }
35        view_port_update(view_port);
36        release_mutex(&state_mutex, plugin_state);
37    }
38    ...

```

---

## Drawing Graphics

Creating graphics on flipper has been made easy by flippers developers. An canvas around the outer edges of the screen could be easily added with a single line: `canvas_draw_frame(canvas, 0, 0, 128, 64);`.

However, when it comes to dealing with user input, moving objects, changing processes we have to take into account that objects might be used by other threads. In the previous part we added a mutex in order to block any other thread writing to an object. For safe drawing graphics, we have to do the same.

1. Acquire a mutex by calling `acquire_mutex()`. The `render_callback()` has the context in a argument. Previously we told the `set_callback` function to use `plugin_state` for this.
2. Check if the mutex is valid, otherwise skip this render
3. Do all the drawing that we like.. In this case a simple text Hello world, on the `x` and `y` positions we have set in the `plugin_state`.
4. Close the mutex again.

```
1 static void render_callback(Canvas* const canvas, void* ctx) {
2     const PluginState* plugin_state = acquire_mutex((ValueMutex*)ctx,
3         25);
4     if(plugin_state == NULL) {
5         return;
6     }
7     // border around the edge of the screen
8     canvas_draw_frame(canvas, 0, 0, 128, 64);
9
10    canvas_set_font(canvas, FontPrimary);
11    canvas_draw_str_aligned(canvas, plugin_state.x, plugin_state.y,
12        AlignRight, AlignBottom, "Hello World");
13    release_mutex((ValueMutex*)ctx, plugin_state);
14 }
```

## Building the firmware + plugin

Before the plugin is added to flipper. We have to let the compiler know, where to find the plugins files.

1. The applications needs a manifest file to let the firmware know how to use it. Therefore create a file `applications\hello_world\application.fam`

```
1 App(
2     appid="hello_world",
```

---

```
3     name="Hello World",
4     apptype=FlipperAppType.PLUGIN,
5     entry_point="hello_world_app",
6     cdefines=["APP_HELLO_WORLD"],
7     requires=[
8         "gui",
9     ],
10    stack_size=2 * 1024,
11    order=20,
12 )
```

2. The application needs to be registered in the menu to be called. This is possible by adding an entry to `applications\meta\application.fam`

Add the application id to the list of provides for the specific menu. In this case under `basic_plugins`:

```
1 App(
2     appid="basic_plugins",
3     name="Basic applications for plug-in menu",
4     apptype=FlipperAppType.METAPACKAGE,
5     provides=[
6         "music_player",
7         "bt_hid",
8         "picopass",
9         "hello_world",
10    ],
11 )
```

Now you can build the application!