Assignment 5

Traffic Light Remake

1 Goal

In classes students learned about how a system can be partitioned into components, and how different components communicate with each other via well-defined event interfaces. In this exercise students put what they learned into practice by hooking up multiple components to form a functional system.

2 Readings

1. Finite State Machines for Integration and Control in ALICE. Giacinto De Cataldo, etc. Proceedings of ICALEPCS07, Knoxville, Tennessee, USA. 2007. (https://accelconf.web.cern.ch/accelconf/ica07/PAPERS/RPPB21.PDF)

   This paper illustrates the use of a hierarchy of finite state machines in ALICE (A Large Ion Collider Experiment) on the Large Hadron Collider (LHC) in CERN. This is one of the many examples of the application of formal state machines in critical systems. Note that in this system each state machine itself is not hierarchical.

2. A Platform Independent Framework for Statecharts Code Generation. L. Andolfato, etc. Proceedings of ICALEPCS. 2011.

   https://accelconf.web.cern.ch/icalepcs2011/papers/weaault03.pdf

   https://accelconf.web.cern.ch/icalepcs2011/talks/weaault03_talk.pdf

   This paper and presentation describes the use of statecharts in the Very Large Telescope (VLT) control system in Chile.

3 Setup

1. Continue on from the previous assignment, Assignment 4 – Traffic Light.

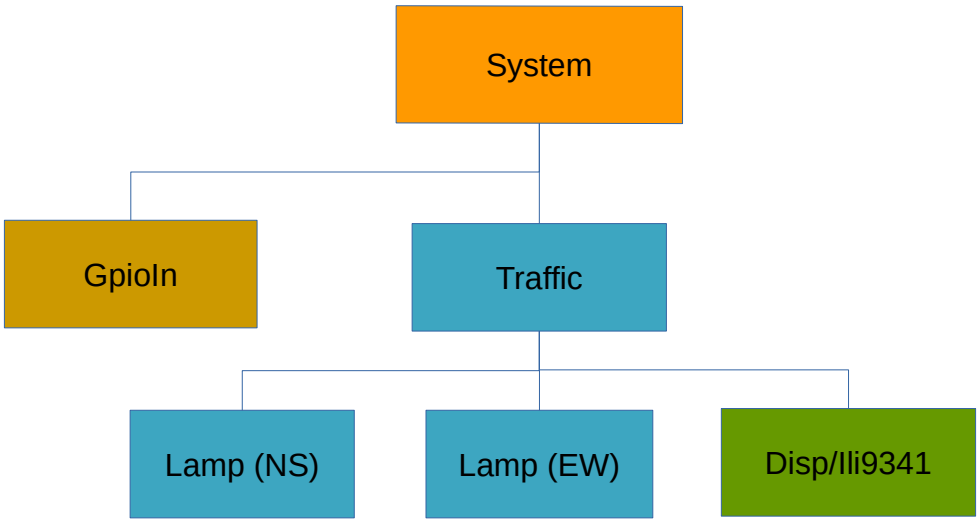
4 Tasks

Our demo system involves the following components:

- System (SYSTEM) - System manager coordinating other components.
- GpioIn (USER_BTN) - GPIO input connected to the USER button.

- Traffic (TRAFFIC) - Traffic light controller with two Lamp (LAMP_NS and LAMP_EW) orthogonal regions.
- Disp/Ili9431 (ILI9341) - Display controller.

The following diagram shows the control hierarchy of these components.



Tasks for this assignment are:

1. In **Src/app/System/System.cpp** *Started* state, observe the handling of GPIO_IN_PULSE_IND and GPIO_IN_HOLD_IND events by sending the TRAFFIC_CAR_NS_REQ and TRAFFIC_CAR_EW_REQ events to **Traffic** respectively.

   This simulates a car arriving along the NS direction with a short press (<1s) on the USER button. It simulates a car arriving along the EW direction with a hold (>1s) on the USER button. If you keep holding the button, it simulates cars keep arriving along the EW direction.

   In this example, the System acts as a coordinator between the GpioIn component and the Traffic component

2. Since **Traffic** now needs to use the display service provided by **Ili9341**, it needs to initialize/start the **Ili9341** active object when it is started.

   First, note that the display event interface header has been added to the top of **Src/app/Traffic/Traffic.cpp**:

   ```
   #include "DispInterface.h"
   ```

   In *Traffic::Stopped* state, upon TRAFFIC_START_REQ, send the events DISP_START_REQ

and DISP_DRAW_BEGIN_REQ to Ili9341 active object:

```
me->Send(new DispStartReq(), ILI9341);
me->Send(new DispDrawBeginReq(), ILI9341);
```

For completeness, perform the reverse action upon TRAFFIC_STOP_REQ in the *Traffic::Started* state by sending the DISP_STOP_REQ to **Ili9341**. Note we can but do not have to send the DISP_DRAW_END_REQ to **Ili9341** before stopping it.

```
me->Send(new DispStopReq(), ILI9341);
```

Note that events for **Ili9341** use the prefix "DISP_" rather than "ILI9341_". This is an attempt to use a common event interface for related HSM classes, so that we don't have to modify all the event names when we switch from one display driver to another.

Having **Traffic** starting and stopping **Ili9341** illustrates the concept of hierarchical control pattern.

3. In **Src/app/Traffic/Lamp/Lamp.cpp**, implement the member function Lamp::Draw(). This is a helper function to draw the graphics of the two traffic lamps on the LCD (one for the NS and one for EW direction). We are going to send the DISP_DRAW_TEXT_REQ and DISP_DRAW_RECT_REQ to **Ili9341** to render the graphics.

   Like before, note that the display event interface header has been added to the top of **Src/app/Traffic/Lamp/Lamp.cpp**:

   ```
   #include "DispInterface.h"
   ```

   Implement Lamp::Draw in Lamp.cpp. A skeleton is provided in the source code.

   ```
   // Helper functions.
   void Lamp::Draw(Hsmn hsmn, bool redOn, bool yellowOn, bool greenOn) {
       // Assignment 5
       (void)hsmn;
       (void)redOn;
       (void)yellowOn;
       (void)greenOn;
   }
   ```

   Hint :

   a) You can draw text and rectangle by sending these events:

      ```
      Send(new DispDrawTextReq(...), ILI9341);
      Send(new DispDrawRectReq(...), ILI9341);
      ```

   b) In Draw(), you need to handle the two *Lamp* instances, namely *LAMP_NS* and *LAMP_EW*, based on the parameter *hsmn*.

4. In **src/Traffic/Lamp/Lamp.cpp**, call the Draw() member function at appropriate places to draw traffic lights on the display module. You can get the *hsmn* of the current Lamp instance with "`me->GetHsmn()`".

5. Test your code by pressing and holding the USER button. Ensure the traffic lights on the LCD displays show the expected patterns. Use log messages in the UART console to debug and verify your application.

   Note – You can control log output with the "**log on/off**" command. You can check component states and component numbers (HSMN) with the "**state**" and "**hsm**" command.