

Punto 1.b

En la figura 1 se puede ver el contenido del archivo *prefix.sct*. Este contiene el diagrama de estados del ejemplo “Blinky” que se utilizó en el Tp1. Este archivo prefix corresponde al que se encuentra en la carpeta *firmware_v2/projects/TP2/statecharts_bare_metal/gen*, clonado de */firmware_v2/sapi_examples/statecharts/statecharts_bare_metal/gen*.

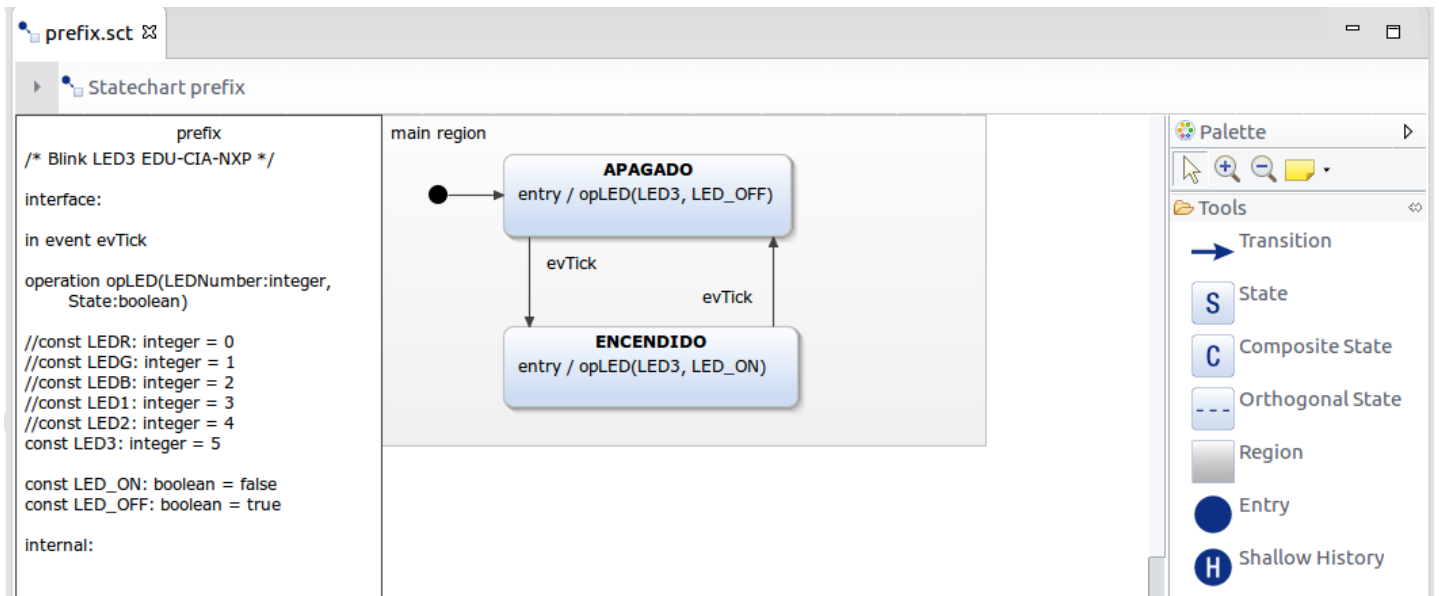


Figura 1: Diagrama de estados del ejemplo “Blinky” del TP1.

Para generar los archivos de código correspondientes al diagrama de estados, se hace click derecho sobre el archivo *prefix.sgen* y se selecciona la opción “Generate Code Artifacts”. Este se puede ver en la figura 2.

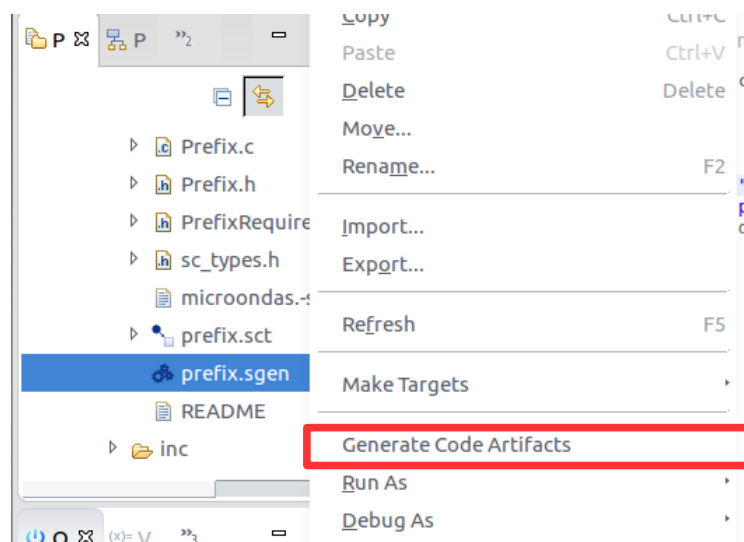


Figura 2: Opción donde se generan los archivos de código del ejemplo del diagrama de estados de la figura 1.

En la figura 3 se muestran los archivos que se generan con esta opción. Estos son *Prefix.c*, *Prefix.h*, *PrefixRequired.h* y *sc_types.h*.

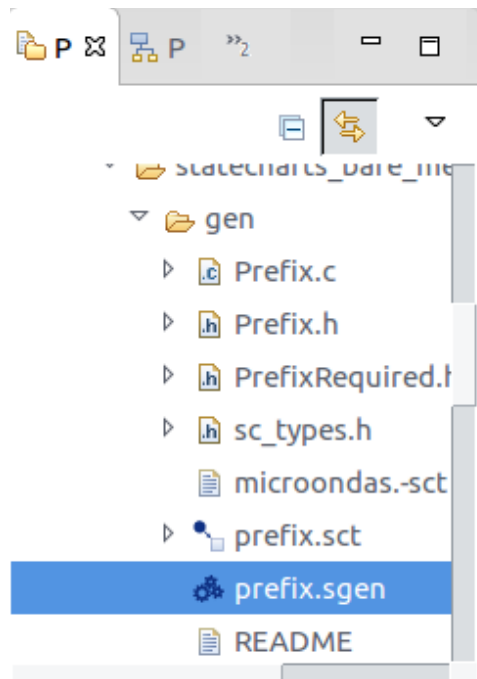


Figura 3: Archivos generados a partir de la opción “Generate Code Artifacts”.

Estos archivos son usados por el archivo *main.c* que se encuentra en *firmware_v2/projects/TP2/statecharts_bare_metal/src*. En la figura 4 se ve la inclusión del archivo *Prefix.h* dentro del archivo *main.c*.

```

1  /* Copyright 2017, Pablo Ridolfi, Juan Esteban Alarcón, Juan Manuel Cruz
33
34  /** @brief This is a simple statechart example using Yakindu Statechart Tool
35   * Plug-in (update site: http://updates.yakindu.org/sct/mars/releases/).
36   */
37
38  /** \addtogroup statechart Simple UML Statechart example.
39   ** @{ */
40
41  /*===== [inclusions] =====*/
42
43  #include "main.h"
44  #include "sapi.h"          // <= SAPI header
45
46  /* Include statechart header file. Be sure you run the statechart C code
47   * generation tool!
48   */
49  #include "Prefix.h"
50  #include "TimerTicks.h"
51
52
53  /*===== [macros and definitions] =====*/
54  /* Compilation choices */
55  #define SCT_1 (1)          /* Test Statechart EDU-CIAA-NXP - Blink LED3
56                               #define __USE_TIME_EVENTS (false)
57                               rm prefix.sct
58                               cp Blink.-sct prefix.sct
59  #define SCT_1 (1)          /* Test Statechart EDU-CIAA-NXP - Blink TimeEvent LED3
60                               #define __USE_TIME_EVENTS (true)
61                               rm prefix.sct
62                               cp BlinkTimeEvent.-sct prefix.sct

```

Figura 4: Archivo *main.c*, donde se muestra que se incluye el archivo *Prefix.h*, el cual fue generado previamente con la opción “Generate Code Artifacts”.

Dentro de este archivo se define una MACRO TEST y dependiendo de su valor se selecciona qué ejemplo se va a ejecutar. Para este caso TEST = SCT_1, es decir que se va a ejecutar el ejemplo del blinky, como se llega a ver en la figura 4. A continuación se analiza el main y su desarrollo, haciendo uso del diagrama de estados creado con Yakindu.

En la figura 5 se puede ver el comienzo del main. Lo primero que se ve es la presencia de un condicional para el compilador. “__USE_TIME_EVENTS” es una macro que se encuentra definida más arriba en el *main.c*. Se puede ver en la figura 6.

```
main.c
196  */
197  int main(void)
198  {
199      #if ( __USE_TIME_EVENTS == true)
200          uint32_t i;
201      #endif
202
203      /* Generic Initialization */
204      boardConfig();
205
206      /* Init Ticks counter => TICKRATE_MS */
207      tickConfig( TICKRATE_MS );
208
209      /* Add Tick Hook */
210      tickCallbackSet( myTickHook, (void*)NULL );
211
212      /* Statechart Initialization */
213      #if ( __USE_TIME_EVENTS == true)
214          InitTimerTicks(ticks, NOF_TIMERS);
215      #endif
216
217      prefix_init(&statechart);
218      prefix_enter(&statechart);
219
220      /* LEDs toggle in main */
```

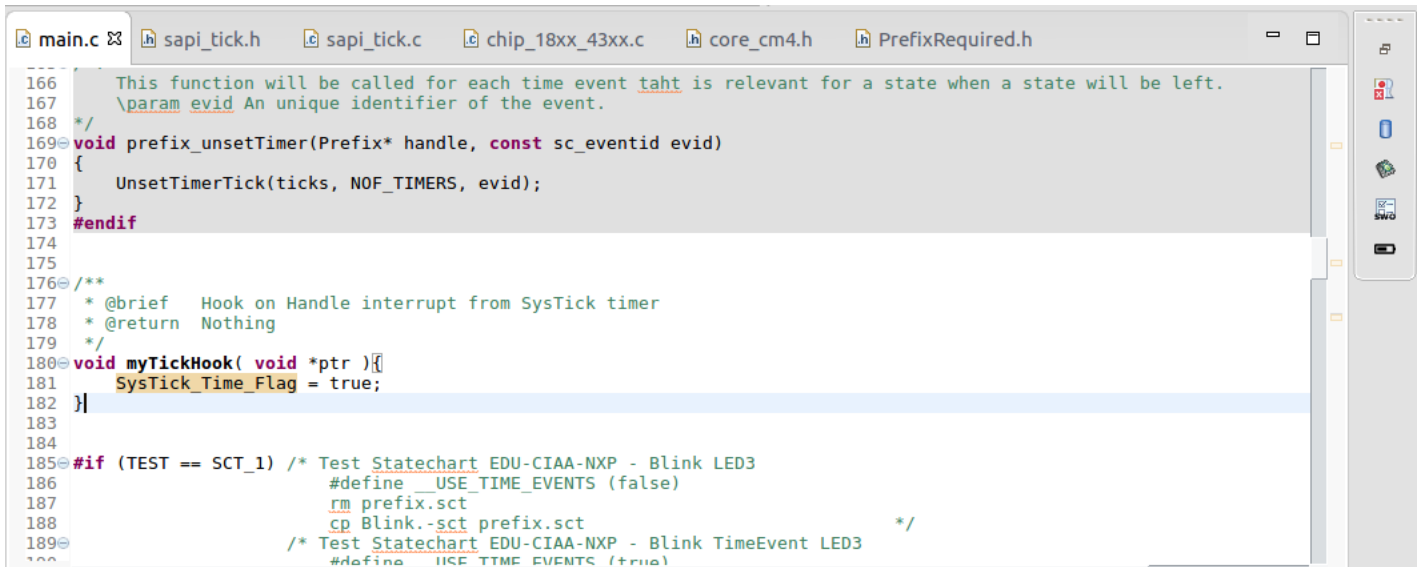
Figura 5: Comienzo del main del ejemplo del diagrama de estados de la figura 1.

```
main.c
84  #define TICKRATE_1MS      (1)           /* 1000 ticks per second */
85  #define TICKRATE_MS      (TICKRATE_1MS) /* 1000 ticks per second */
86  /*=====[internal data declaration]=====*/
87
88  volatile bool SysTick_Time_Flag = false;
89
90  /*! This is a state machine */
91  static Prefix statechart;
92
93
94  /* Select a TimeEvents choose */
95  #define __USE_TIME_EVENTS (false) /* "false" without TimeEvents */
96  // #define __USE_TIME_EVENTS (true) /* or "true" with TimerEvents */
97
98  /*! This is a timed state machine that requires timer services */
99  #if ( __USE_TIME_EVENTS == true)
100      #define NOF_TIMERS (sizeof(PrefixTimeEvents)/sizeof(sc_boolean))
101  #else
102      #define NOF_TIMERS 0
103  #endif
104
105  TimerTicks ticks[NOF_TIMERS];
106
107
```

Figura 6: Definición de la MACRO __USE_TIME_EVENTS. Se ve que se encuentra en false.

La MACRO mencionada sirve para el uso de eventos del statechart, triggereados por el cumplimiento de un timer. Es decir, que el diagrama de estados cambiaría de estado automáticamente una vez que se cumpla un determinado tiempo. En este ejemplo uno tendería a decir que debería estar en “true”, ya que una vez cumplido el tiempo seteado, el LED se prende o se apagara, es decir, cambia de estado. Para ver porqué sucede esto, se analizan las funciones utilizadas en el main.

Lo siguiente que se hace en la figura 5 es configurar el tickrate y el tickcallbackset, es decir, el tiempo de interrupción y la función que se invoca en esa interrupción. Esta función se encuentra definida en el main y hace lo que se muestra en la figura 7.

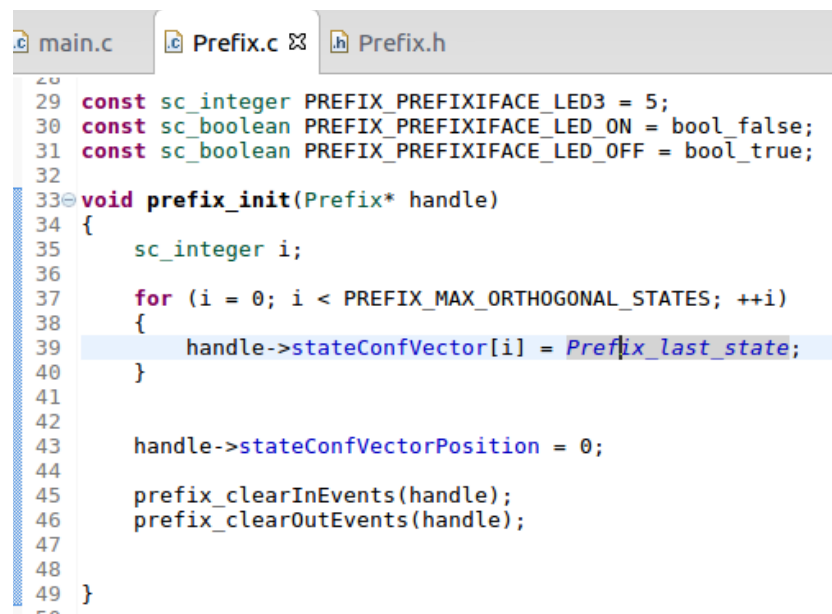


```
166 This function will be called for each time event taht is relevant for a state when a state will be left.
167 \param evid An unique identifier of the event.
168 */
169 void prefix_unsetTimer(Prefix* handle, const sc_eventid evid)
170 {
171     UnsetTimerTick(ticks, NOF_TIMERS, evid);
172 }
173 #endif
174
175
176 /**
177  * @brief Hook on Handle interrupt from SysTick timer
178  * @return Nothing
179  */
180 void myTickHook( void *ptr ){
181     SysTick_Time_Flag = true;
182 }
183
184
185 #if (TEST == SCT_1) /* Test Statechart EDU-CIAA-NXP - Blink LED3
186                     #define __USE_TIME_EVENTS (false)
187                     rm prefix.sct
188                     cp Blink-.sct prefix.sct
189                     /* Test Statechart EDU-CIAA-NXP - Blink TimeEvent LED3
190                     #define __USE_TIME_EVENTS (true)
```

Figura 7: Función “myTickHook” que se invoca en la interrupción, definida en el *main.c*.

La función solamente setea una variable, definida en el *main.c*, en true.

Continuando con la figura 5, se invoca a *prefix_init* y *prefix_enter*. Ambas funciones reciben la dirección de memoria de una variable “statechart”, definida en el *main.c*. Empezando por la función “*prefix_init*”, esta puede verse en la figura 8.



```
29 const sc_integer PREFIX_PREFIXIFACE_LED3 = 5;
30 const sc_boolean PREFIX_PREFIXIFACE_LED_ON = bool_false;
31 const sc_boolean PREFIX_PREFIXIFACE_LED_OFF = bool_true;
32
33 void prefix_init(Prefix* handle)
34 {
35     sc_integer i;
36
37     for (i = 0; i < PREFIX_MAX_ORTHOGONAL_STATES; ++i)
38     {
39         handle->stateConfVector[i] = Prefix_last_state;
40     }
41
42     handle->stateConfVectorPosition = 0;
43
44     prefix_clearInEvents(handle);
45     prefix_clearOutEvents(handle);
46
47 }
48
49 }
```

Figura 8: Función “*prefix_init*”, que se invoca en el main.

La variable “i” del tipo *sc_integer* es equivalente a una variable del tipo “*int32_t*”, simplemente las llama de otra forma para saber que son internas del statechart. Luego hay un ciclo **for**, donde se compara la variable “i” con una macro *PREFIX_MAX_ORTHOGONAL_STATES*, que se encuentra definida en *Prefix.h* y para este caso vale 1. Estos Orthogonal States, según la documentación de Yakindu, son:

An **orthogonal state** is basically a [composite state](#) with more than one region. These regions are executed virtually concurrently. **Please note** the word *virtually*! YAKINDU Statechart Tools does not guarantee in any way that orthogonal regions are *really* executed concurrently. At the moment, no code generator utilizes threads to achieve this. Orthogonal states should rather be understood as a manner to have two or more sub-statecharts working together, however, they are executed *one after the other* in every cycle, and in a defined order: top to bottom, left to right. The same applies to multiple regions in the statechart itself. Please consult section ["Raising and processing an event"](#) for further information on region priorities and their meanings for the statechart execution.

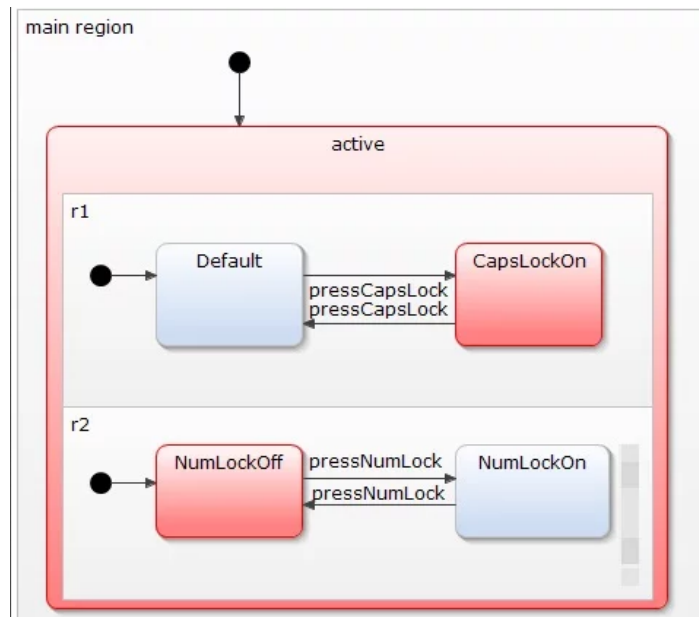


Figura 9: Ejemplo de Orthogonal States, extraído de la documentación de Yakindu.

Como se hace una iteración sola, se accede al parámetro `stateConfVector[1]` (este es un elemento de la variable `handle` del tipo `Prefix`), que es un vector de tamaño `PREFIX_MAX_ORTHOGONAL_STATES` (en este caso 1) y se le asigna el valor `Prefix_last_state`. Esto lo que quiere decir es que en este paso se setean todos los Orthogonal States en el estado inicial que corresponde al punto negro de la figura 1. En este caso hay uno solo, así que solo se setea este. Luego se setea el parámetro `stateConfVectorPosition` en 0. Por último se invoca a `“prefix_clearInevents”` pasándole como parámetro el mismo `“handle”`. Esta, como se muestra en la figura 10, accede al elemento `“iface”`, que a su vez es otra estructura del tipo `“PrefixIface”`, que solamente contiene un elemento del tipo `“sc_boolean”` que puede valer `true` o `false`. En esta función se la está seteando en `“false”`. La función `“prefix_clearOutEvents”` en este caso no cumple ninguna función.

Continuando con el main, se analiza la función `“prefix_enter”`, que también recibe la dirección de memoria del statechart. El objetivo de esta función es entrar a un estado que no sea el estado inicial, dentro del statechart. La función se puede ver en la figura 11,

```

84
85 static void prefix_clearInEvents(Prefix* handle)
86 {
87     handle->iface.evTick_raised = bool_false;
88 }
89

```

Figura 10: Función `“prefix_clearInEvents”`. Esta setea el elemento `evTick_raised` en `“false”`, indicando que el evento que acciona al diagrama de estados no ocurrió aún.

```

50
51 void prefix_enter(Prefix* handle)
52 {
53     /* Default enter sequence for statechart prefix */
54     prefix_enseq_main_region_default(handle);
55 }
56

```

Figura 11: Función “prefix_enter” que se invoca en el main.

la cual a su vez invoca a otra función “prefix_enseq_main_region_default” que recibe el parámetro “handle”. Esta función corresponde a la secuencia de entrada por defecto, al statechart. Dentro de esta a su vez llama a otra función “prefix_react_main_region__entry_Default” que también recibe la función “handle”. El trabajo de esta es realizar el evento de entrar al primer estado, el cual, según la figura 1, correspondería con tener el LED apagado. Es entonces que esta función invoca a una última función más “prefix_enseq_main_region_APAGADO_default” que también recibe el parámetro handle, cuyo objetivo es entrar al estado apagado. Esta se puede ver en la figura 12. A continuación se analiza las funciones invocadas.

```

202 /* 'default' enter sequence for state APAGADO */
203 static void prefix_enseq_main_region_APAGADO_default(Prefix* handle)
204 {
205     /* 'default' enter sequence for state APAGADO */
206     prefix_enact_main_region_APAGADO(handle);
207     handle->stateConfVector[0] = Prefix_main_region_APAGADO;
208     handle->stateConfVectorPosition = 0;
209 }
210

```

Figura 12: Contenido de la función “prefix_enseq_main_region_APAGADO_default”, definida en el archivo *Prefix.c*.

Primero se llama a una función “prefix_enact_main_region_APAGADO” que recibe el handler. Esta se puede ver en la figura 13 y lo que hace es realizar la acción correspondiente al estado “APAGADO”, es decir, invoca a la función que apaga el LED.

```

188 /* Entry action for state 'APAGADO'. */
189 static void prefix_enact_main_region_APAGADO(Prefix* handle)
190 {
191     /* Entry action for state 'APAGADO'. */
192     prefixIface_opLED(handle, PREFIX_PREFIXIFACE_LED3, PREFIX_PREFIXIFACE_LED_OFF);
193 }
194

```

Figura 13: Función “prefix_enact_main_region_APAGADO”. Corresponde a la realización de la acción que se realiza dentro del estado APAGADO. Invoca a la función “prefixIface_opLED” que apaga el LED.

Continuando con la función “prefix_enseq_main_region_APAGADO_default”, esta setea el estado del “stateConfVector” en “Prefix_main_region_APAGADO”, lo cual indica que ahora el statechart se encuentra en el estado “APAGADO” (recordar que el vector stateConfVector contiene un elemento por cada Orthogonal State, en este caso el vector stateConfVector tiene un solo elemento ya que solo se tiene un Orthogonal State). Por último se setea el valor del elemento “stateConfVectorPosition” en 0. Cabe aclarar que la función “prefixIface_opLED” se encuentra declarada en el archivo “PrefixRequired.h”, archivo que se generó automáticamente por el Yakinidu. Las declaraciones en esta función me piden que las defina en otro archivo, manualmente.

Con el análisis realizado, puede analizarse el funcionamiento general del main. Como se ve en la figura 14, luego de las funciones descritas se entra en un **while(1)**.

```

main.c Prefix.c Prefix.h sc_types.h PrefixRequired.h
220 // LEDS toggle in main //
221 while (1) {
222     __WFI();
223
224     if (SysTick_Time_Flag == true) {
225         SysTick_Time_Flag = false;
226
227         #if (__USE_TIME_EVENTS == true)
228             UpdateTimers(ticks, NOF_TIMERS);
229             for (i = 0; i < NOF_TIMERS; i++) {
230                 if (IsPendEvent(ticks, NOF_TIMERS, ticks[i].evid) == true) {
231
232                     prefix_raiseTimeEvent(&statechart, ticks[i].evid); // Event -> Ticks.evid => OK
233                     MarkAsAttEvent(ticks, NOF_TIMERS, ticks[i].evid);
234                 }
235             }
236         #else
237             prefixIface_raise_evTick(&statechart); // Event -> evTick => OK
238         #endif
239
240         prefix_runCycle(&statechart); // Run Cycle of Statechart
241         //delay(100);
242     }
243 }
244 }

```

Figura 14: Continuación del main del ejemplo del Blinky.

Respecto a la función “__WFI()”, según la documentación de esta, corresponde a una función que suspende la ejecución del main hasta que encuentre una interrupción. En este programa la interrupción correspondería al caso del “Tick”. Cuando se cumple el tiempo seteado para la interrupción del Tick, se invoca a “myTickHook”, que como ya se dijo, setea una variable en true. Luego el main detecta que esto sucedió y entra en el **if**. Vuelve a setear la variable en false y llama a dos funciones: “prefixIface_raise_evTick” y a “prefix_runCycle”. La primera se ve en la figura 15 y su objetivo es indicarle al statechart que se ejecutó el evento evTick.

```

144
145 void prefixIface_raise_evTick(Prefix* handle)
146 {
147     handle->iface.evTick_raised = bool_true;
148 }
149

```

Figura 14: Función que indica al statechart que sucedió el evento “evTick”.

Se modifica el elemento “iface” del handle, accediendo al evento “evTick_raised” indicándole que se activó. Por último se invoca a la función “prefix_runCycle”. Esta se puede ver en la figura 15.


```

94 void prefix_runCycle(Prefix* handle)
95 {
96
97     prefix_clearOutEvents(handle);
98
99     for (handle->stateConfVectorPosition = 0;
100         handle->stateConfVectorPosition < PREFIX_MAX_ORTHOGONAL_STATES;
101         handle->stateConfVectorPosition++)
102     {
103
104         switch (handle->stateConfVector[handle->stateConfVectorPosition])
105         {
106             case Prefix_main_region_APAGADO :
107             {
108                 prefix_react_main_region_APAGADO(handle);
109                 break;
110             }
111             case Prefix_main_region_ENCENDIDO :
112             {
113                 prefix_react_main_region_ENCENDIDO(handle);
114                 break;
115             }
116             default:
117                 break;
118         }
119     }
120
121     prefix_clearInEvents(handle);
122 }
123

```

Figura 15: Función “prefix_runCycle” que se invoca en el main.

Primero se invoca a “prefix_clearOutEvents” que como ya se dijo en este caso no hace nada. Luego el ciclo **for** se repite una cantidad de veces igual a la cantidad de Orthogonal States que se tienen, que en este caso es 1. Lo que se hace dentro del ciclo es setear el elemento “stateConfVectorPosition” e incrementarlo en 1 en cada iteración. Dentro del ciclo hay un **switch**. La variable a evaluar es la posición “stateConfVectorPosition” (que ahora vale 0) del vector “stateConfVector”. Se refiere a que se va a trabajar en el único Orthogonal State que se tiene en este ejemplo. Como se había hecho antes, la variable que corresponde a la posición 0 del vector “stateConfVector” tiene el valor “Prefix_main_region_APAGADO”, entonces se entra al case correspondiente y se invoca a la función “prefix_react_main_region_APAGADO”. La función esta se puede ver en la figura 16.

```

256 {
257     prefix_exseq_main_region_ENCENDIDO(handle);
258     break;
259 }
260 default: break;
261 }
262 }
263
264 /* The reactions of state APAGADO. */
265 static void prefix_react_main_region_APAGADO(Prefix* handle)
266 {
267     /* The reactions of state APAGADO. */
268     if (prefix_check_main_region_APAGADO_tr0_tr0(handle) == bool_true)
269     {
270         prefix_effect_main_region_APAGADO_tr0(handle);
271     }
272 }
273

```

Figura 16: Función “prefix_react_main_region_APAGADO”, chequea si el elemento “evTickraised” del elemento “iface” de la estructura “handle” es true. En caso de serlo ejecuta una reacción.

La función “prefix_check_main_region_APAGADO_tr0_tr0” simplemente chequea el elemento “evTickraised” del elemento “iface” de la estructura “handle” para ver si es true, lo cual quiere decir que sucedió el evento que triggerea el cambio de estado. En este caso esto es cierto, ya que se hizo previamente con la función de la figura 14. Luego se ejecuta “prefix_effect_main_region_APAGADO_tr0” la cual ejecuta la reacción, es decir, el cambio de estado. Dentro de esta se ejecuta la función “prefix_exseq_main_region_APAGADO”, función que solamente deja el stateConfVector[0] en el estado inicial del puntito negro y deja el contador stateConfVectorPosition = 0. Por último se ejecuta la secuencia de entrada al estado “ENCENDIDO” con la función “prefix_enseq_main_region_ENCENDIDO” que solamente ejecuta la acción de encender el LED y setear el “stateConfVector[0] en el estado “ENCENDIDO”. Esta última función “prefix_enseq_main_region_ENCENDIDO” es de la misma forma que la función de la figura 12, pero para el estado ENCENDIDO.

Volviendo a la figura 15, al final de dicha función se ejecuta “prefix_clearInEvents”, función que se analizó en la figura 10 y solo le dice al statechart que el evento que triggerea el cambio de estados no está sucediendo.

Al principio se dijo que se estaba utilizando un evento por cumplimiento de un timer y que por esto, la MACRO “__USE_TIME_EVENTS” debería estar en “true” pero se encuentra en “false”. Entonces, a partir de lo visto hasta acá, sucede que, cuando se cumple el timer y se invoca a la interrupción de Ticks, se está levantando un flag, el cual luego es analizado en el main. Es decir, no se hace el cambio de estado por un timer si no por el valor del flag, si es “true” o “false”.