10.7 Preemptive scheduler

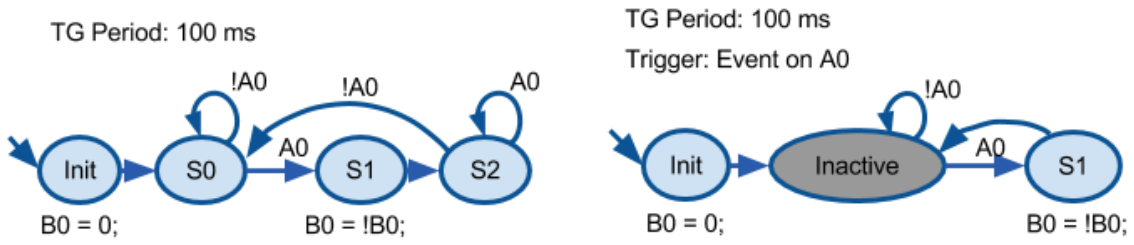
10.8 Triggered synchSMs

Sometimes a synchSM tick just samples an input (such as input A0) to detect a change, a process known as ***polling***. To reduce the microcontroller being used for polling, some microcontrollers come with special hardware that detects a change on an input and calls a special ISR in response such as inputChangeISR(). Reducing polling reduces microcontroller utilization, decreases jitter, reduces missed deadlines, and/or enables more tasks to be implemented on the microcontroller.

Availability of such hardware inspires a variation of a synchSM that can make itself inactive by transitioning to a special "Inactive" state. *While inactive, a synchSM does not tick* and thus when implemented will not utilize the microcontroller. The synchSM becomes active again when a specified event occurs, such as an event on A0, after which the synchSM ticks repeatedly at its normal period, until transitioning to the inactive state again. The event triggers the synchSM to become active, and thus such a synchSM is called a ***triggered synchSM***.

Consider a system that toggles B0 whenever a button connected to A0 is pressed. One way to capture this behavior uses a synchSM that polls A0 every 100 ms, as shown on the left below.

Figure 10.8.1: Replacing a polling synchSM (left) by a triggered synchSM (right).



An alternative using a triggered synchSM is shown on the right. After initializing B0, the synchSM becomes inactive. While inactive, the synchSM *does not tick*. When an event occurs on A0, the synchSM becomes active again and checks transitions leaving the Inactive state. A0 being 1 means the event was a change from 0 to 1, so the synchSM goes to state S1, which toggles B0, after which the synchSM transitions back to the Inactive state. Instead, A0 being 0 means the event was a change from 1 to 0, which should not change B0, and thus the transition goes back to the Inactive state. Detecting the event on A0 is not the responsibility of the synchSM; the detection occurs by some other means outside the synchSM.

In scheduling terminology, a task triggered by an event is known as an ***aperiodic*** task, in contrast to a ***periodic*** task that ticks at a known rate.

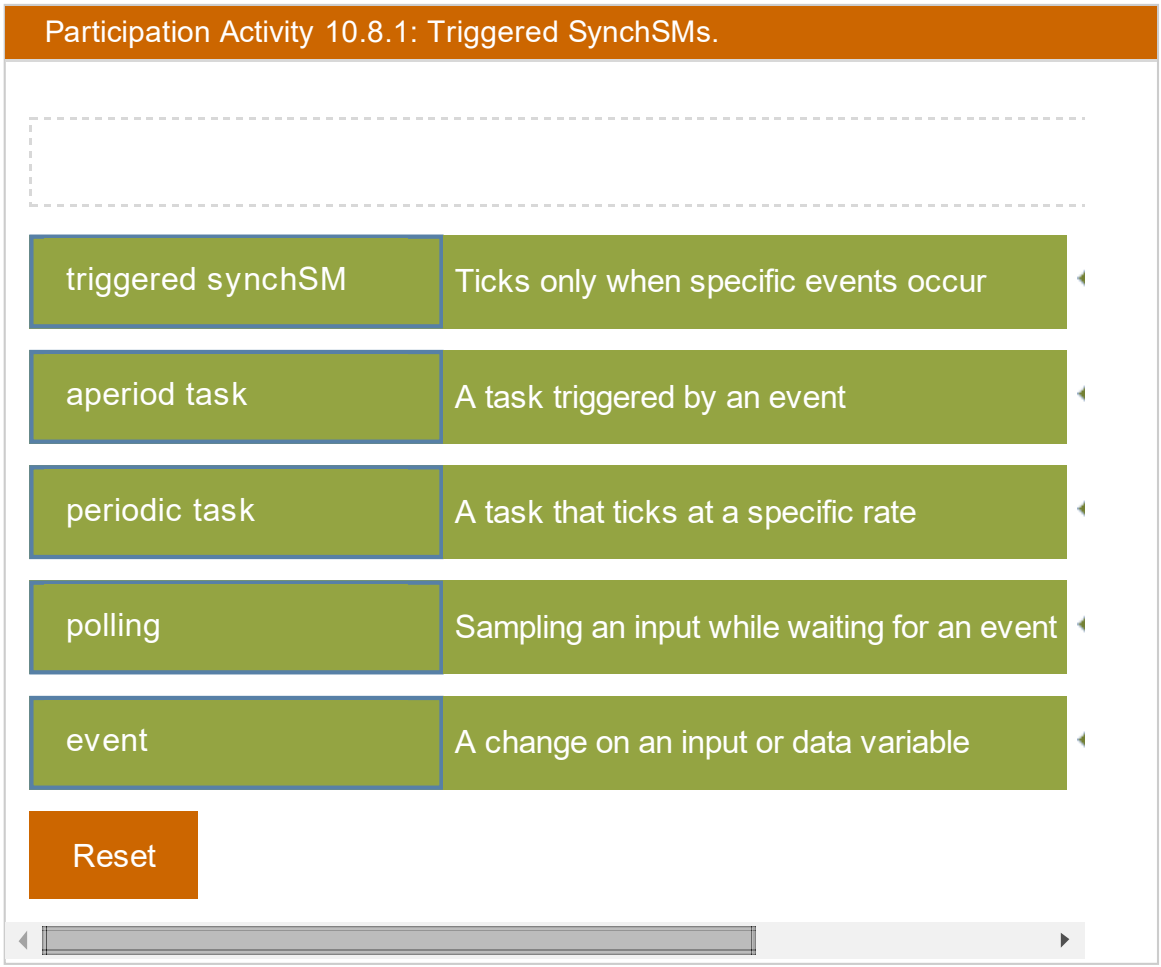
A benefit of a triggered synchSM becomes apparent when examining the utilization of a microcontroller implementing the synchSM, where that microcontroller has special hardware to detect the triggering event.

Figure 10.8.2: Triggered synchSMs can reduce utilization.

The timing diagram on the left shows TG with polling, executing every 100 ms. The ticks for state S0 poll the input A0 looking for a change from 0 to 1, and the ticks for state S2 for a change from 1 to 0. The timing diagram on the right shows a triggered TG. TG goes inactive while waiting for an event on A0, and does not execute during that time, reducing microcontroller utilization.

Using existing special hardware that detects events on pins can yield the additional benefit of sampling the pin faster than sampling achieved by polling done by a task on a microprocessor.

Participation Activity 10.8.1: Triggered SynchSMs.



A triggered synchSM can be implemented in C on a microcontroller with some extensions to the earlier scheduler. (Note: RIBS does not currently have support for triggered SMs). RIMS does not currently have built-in support for detecting events on pins, but RIMS does have built-in support for detecting a UART character receive, and thus a UART receive can be used to trigger a synchSM. Thus we shall introduce an example having a synchSM triggered by a UART receive. Consider a modification of an earlier example (that was used to demonstrate scheduler code) where one task blinks B0 repeatedly, and another task lights three LEDs B5, B6, B7 in sequence once whenever the letter 'g' is received by the UART:

Figure 10.8.3: BlinkLed synchSM.

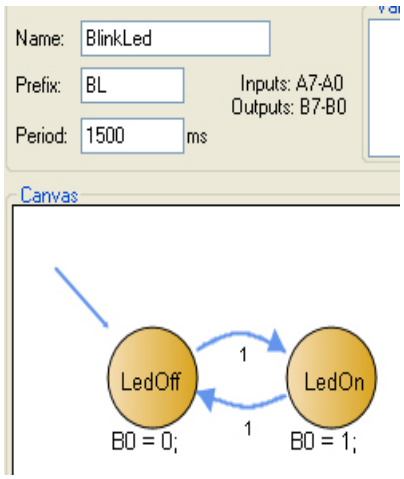
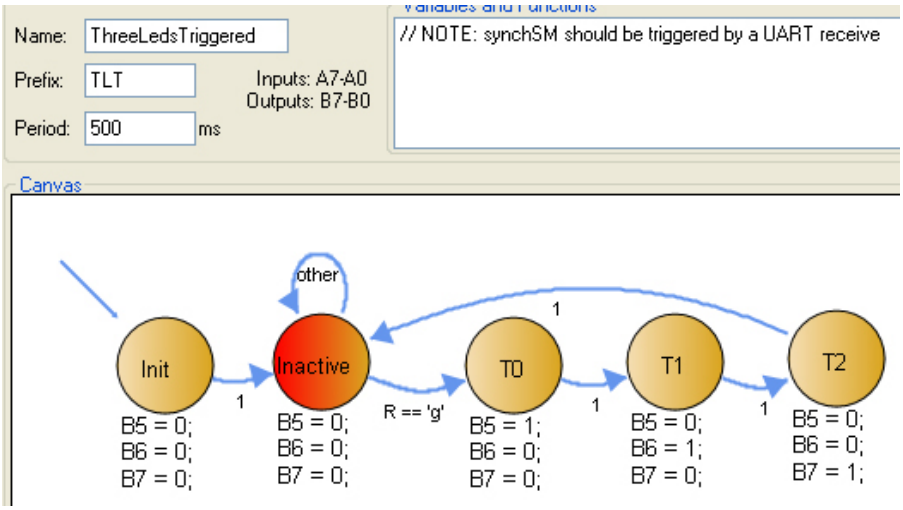


Figure 10.8.4: ThreeLedsTriggered synchSM.



The following code modifies the earlier scheduler code to support the triggered synchSM. Key changes are highlighted and described in the comments.

Figure 10.8.5: Scheduler modified for triggered synchSMs.

```
#include "RIMS.h"

typedef struct task {
   signed   char state;
   unsigned long period;
   unsigned long elapsedTime;
   unsigned char active; // 1: active, 0: inactive
   int (*TickFct)(int);
} task;

task tasks[2]; // Global so visible to tick fct and ISR
const unsigned short tasksNum = 2;
```

```
const unsigned long BL_period = 1500;
const unsigned long TLT_period = 500;
const unsigned long tasksPeriodGCD = 500;

enum BL_States { BL_SMStart, BL_LEDOff, BL_LEDOn };
int TickFct_BlinkLed(int state);

enum TLT_States { TLT_SMStart, TLT_T0, TLT_T1, TLT_T2, TLT_Init, TLT_Inactive };
int TickFct_ThreeLedsTriggered(int state);

unsigned char processingRdyTasks = 0;
void TimerISR() {
   unsigned char i;
   if (processingRdyTasks) {
      printf("Timer ticked before task processing done.\r\n");
   }
   else { // Heart of the scheduler code
      processingRdyTasks = 1;
      for (i=0; i < tasksNum; ++i) {
         if (tasks[i].elapsedTime >= tasks[i].period && tasks[i].active) {
         // Ready
            tasks[i].state = tasks[i].TickFct(tasks[i].state);
            tasks[i].elapsedTime = 0;
         }
         if (tasks[i].active) { tasks[i].elapsedTime += tasksPeriodGCD; }
      }
      processingRdyTasks = 0;
   }
}


volatile unsigned char RxFlag=0;
volatile unsigned char RxData=0;

const unsigned int tasksTLTIndex = 1; // Reference the TLT task in RxISR and TickF

void RxISR(void) {
   tasks[tasksTLTIndex].active = 1; // TLT triggered by UART rx; make active
   tasks[tasksTLTIndex].elapsedTime = tasks[tasksTLTIndex].period; // Make ready t
   RxData = R; // Read to global var to avoid problems if chars come too fast
   RxFlag = 1;
}

// Task tick functions

int TickFct_BlinkLed(int state) {
   puts("BL Tick\r\n");
   switch(state) { // Transitions
      case BL_SMStart:
         state = BL_LEDOff;
         break;
      case BL_LEDOff:
         state = BL_LEDOn;
         break;
      case BL_LEDOn:
         state = BL_LEDOff;
         break;
      default:
         state = BL_SMStart;
         break;
   }

   switch(state) { // State actions
      case BL_LEDOff:
         B0 = 0;
```

```
            B0 = 0;
            break;
         case BL_LEDOn:
            B0 = 1;
            break;
         default:
            break;
      }
      return state;
}

int TickFct_ThreeLedsTriggered(int state) {
  puts("TLT Tick\r\n");
   switch(state) { // Transitions
      case TLT_SMStart:
         state = TLT_Init;
         break;
         case TLT_T0:
         if (1) {
            state = TLT_T1;
         }
         break;
      case TLT_T1:
         if (1) {
            state = TLT_T2;
         }
         break;
      case TLT_T2:
         if (1) {
            state = TLT_Inactive;
         }
         break;
      case TLT_Init:
         if (1) {
            state = TLT_Inactive;
         }
         break;
      case TLT_Inactive: // Note: special "Inactive" state's
                         // transitions are same as any other state
         if (RxData == 'g') {
            state = TLT_T0;
         }
         else {
            state = TLT_Inactive;
         }
         break;
      default:
         state = TLT_SMStart;
        break;
   } // Transitions

   switch(state) { // State actions
      case TLT_T0:
         B5 = 1; B6 = 0; B7 = 0;
         break;
      case TLT_T1:
         B5 = 0; B6 = 1; B7 = 0;
         break;
      case TLT_T2:
         B5 = 0; B6 = 0; B7 = 1;
         break;
      case TLT_Init:
         B5 = 0; B6 = 0; B7 = 0;
         break;
      case TLT_Inactive:
         B5 = 0; B6 = 0; B7 = 0;
```

```
            tasks[tasksTLTIndex].active = 0; // Special "Inactive" state
                                             // makes task inactive
            break;
         default:
            break;
      } // State actions
      return state;
}

void main()
{
   unsigned char i=0;
   tasks[i].state       = BL_SMStart;
   tasks[i].period      = BL_period;
   tasks[i].elapsedTime = tasks[i].period;
   tasks[i].active      = 1; // Task starts as active
   tasks[i].TickFct     = &TickFct_BlinkLed;
   i++;
   tasks[i].state       = TLT_SMStart;
   tasks[i].period      = TLT_period;
   tasks[i].elapsedTime = tasks[i].period;
   tasks[i].active      = 1; // Task starts as active
   tasks[i].TickFct     = &TickFct_ThreeLedsTriggered;

   TimerSet(tasksPeriodGCD);
   TimerOn();
   UARTOn();

   while(1) { Sleep(); }
}
```

The main change is the addition of the "active" field to the task structure, 1 meaning active. The initialization in main sets each task initially active. Transitioning to the special "Inactive" state in task TLT's tick function sets the task to inactive. The scheduler skips any inactive tasks when checking if tasks are ready to tick. The UART's ISR "RxISR" re-activates the TLT task, and also sets the task's elapsedTime such that the task is ready to tick. Note that the code moved the task declarations to be global, so that the RxISR and TickFct_ThreeLedsTriggered functions could set the task's active field. Note also at the end of the scheduler code in main that the scheduler's wait can be completed not just by the next timer tick, but also by a UART receive. A global constant "tasksTLTIndex" is declared at global scope, because the index of the TLT task in the tasks array is required by both the RxISR and the TickFct_ThreeLedsTriggered functions.

Try 10.8.1: Run triggered SM in RIMS.

Run the above code in RIMS. Each tick function prints its name when it ticks; note that the TLT task stops ticking, and only the BL task ticks. Now enter an 'a' into RIMS' "UART input" text box. Note that the TLT task ticks once, but because the input was not 'g', transitions back to the Inactive state and thus doesn't tick. Now type 'g' and note that TLT ticks several times (and LEDS B5, B6, B7 blink in sequence), and then again stops ticking .

A triggered synchSM can be implemented on a microcontroller that doesn't have special hardware to detect the triggering event. In that case, when translating to C, we introduce a helper synchSM that polls the appropriate input for the triggering event, and communicates with the triggered synchSM by setting a global flag variable to 1, holding the 1 long enough for the original synchSM to detect it. The original synchSM's inactive state needs a new transition that goes back to the inactive state when the flag is 0, while the existing transitions need to check that the flag is 1. Note that such an implementation does not achieve the benefits of reduced polling or faster sampling, but may lead to more intuitive initial synchSMs. Capturing behavior using triggered synchSMs may also lead to more efficient implementations if the synchSMs are ported to another microcontroller that does have the special hardware.

10.9 Reducing power with sleep