Programmation de FSM en C

MASTER STIC EEA - Module I1

novembre 2007

Résumé

Ce TD étudie différentes façons de coder des automates en C.

1 FSM : codage par tables

Voir le cours.

2 FSM : codage avec `switch`

On veut réaliser la commande d'un système d'asservissement de vitesse avec un moteur pas à pas. Les changements de vitesse doivent se faire en respectant des rampes d'accélération et de décélération.

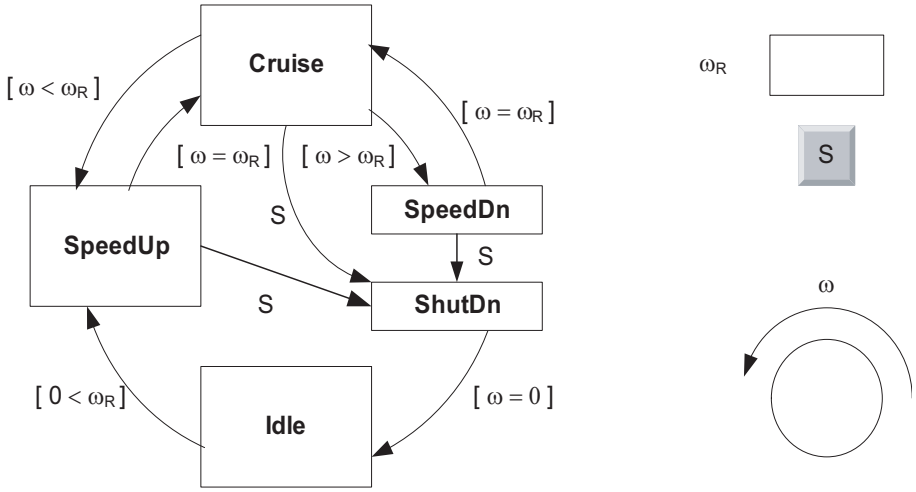

FIG. 1 – Asservissement de vitesse.

La figure 1 représente dans la partie gauche le graphe d'état de la commande. ω est la vitesse mesurée, ω_R est la vitesse de consigne. Le bouton poussoir S demande l'arrêt du moteur. Les comparaisons ($\omega < \omega_R$, $\omega = \omega_R$, ...) sont faites avec une tolérance fixée θ, c'est à dire que $\omega = \omega_R$ signifie $\omega = \omega_R \pm \theta$.

La relation entre la vitesse de rotation ω et la période des impulsions à envoyer au moteur pas-à-pas est définie dans la table `rot2time[]`.

Question 1: Analyser et compléter le code suivant:

```
/*
  file: stepper.c
```

```
  author: C. André
  date: November 24, 2004
  purpose: control of a stepper
*/

#include "RTLclock.h"
#include "bool.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define theta 1

typedef enum {Idle, SpeedUp, SpeedDn, ShutDn, Cruise} State_t;

time_ms_t rot2time[256]; // speed to period table

/* interfacing functions. Externally defined */
int readSensorOmega(); //actual speed reading
int readSensorOmegaR(); // reference speed reading
void step(void); // send a pulse to the stepper
Boolean getS(void); // 1 iff push-button S is pressed

void main()
{
  time_ms_t now, nextTime;
  State_t state = Idle;
  int omega, omegaR, index=0; // 0..255
  Boolean S, preS=false; // previous value of S
  Boolean RES; // event: rising edge on S

  now = currentTime(); // declared in RTLclock.h

  while (1) {
    // acquisitions
    omega = readSensorOmega();
    omegaR = readSensorOmegaR();
    S = getS(); // Boolean value
    RES = S && !preS;
    preS = S;
    // reaction
    switch (state) {
    case Idle:
      if ( omegaR >= theta ) {
          index = 1;
          state = SpeedUp;
      }
      break;
    case SpeedUp:
      if ( RES ) state = ShutDn;
      else
        if ( omega >= omegaR + theta ) state = Cruise;
        else
          if ( index <= omegaR ) index++;
      break;
    case Cruise:
      // provide code
    case ShutDn:
      // provide code
```

```
    case SpeedDn:
      // provide code
    }
    nextTime = now + rot2time[index];
    while ((now = currentTime()) < nextTime);
    if ( index ) step();
  }
}
```

3 FSM: états avec actions d'entrée et de sortie

Code pour la programmation du micro-onde (à analyser).

4 microwave.h

```
/* microwave.h */
/* Charles André */
/* January 4, 2000 */
/* December 17, 2000:  adaptation to the new distribution */

#ifndef _MICROWAVE_H_
#define _MICROWAVE_H_

#ifdef GLOBAL_MICROWAVE_
#define EXT_MICROWAVE_
#else
#define EXT_MICROWAVE_ extern
#endif


#define ON 1
#define OFF 0
/* boolean, true and false are defined in RTLmisc.h */
#include "RTLmisc.h"

#include "RTLclock.h"
#include "RTLtimer.h"


typedef enum {evNone, evFlameON, evCookingSelected,
  evCancel, evDone, evDoorOpens, evTmDD, evTmUD,
  evEnd} Events;

enum timers {timerCD, timerUD, timerDD, timerEnd};

/* global variables */


EXT_MICROWAVE_ boolean present[evEnd];
EXT_MICROWAVE_ void (* nextState) (void);
EXT_MICROWAVE_  time_ms_t cd;
EXT_MICROWAVE_  time_ms_t ud;
EXT_MICROWAVE_  time_ms_t dd;

void initializeState(void);

/* event management */
```

```
void resetEvents(void);
void setEvents(void);

#endif _MICROWAVE_H_
```

5 microw.c

```
/* microw.c */
/* Check microwave implementation */

/* Charles André */
/* January 4, 2000 */
/* December 17, 2000: new environment */

/*
  Use:

  bic -- -appl microw -n 1
  make
  proto -- -appl microw -dis w -re &
  trymw
*/


#include "registerMgr.h"
/* facility for interfacing C/Tcl-Tk proto */
#include "connectMgr.h"
#include "RTLclock.h"
#include "RTLtimer.h"
#include "microwave.h"

#include <stdio.h>
#include <string.h>
#include <assert.h>

/* global variables */


#include "microwCfg.c"

/* debugging facility */

/* print out "Debug:msg" to stderr */
void debugMsg(const char *msg)
{
#ifndef NDEBUG
  fprintf(stderr,"Debug:%s\n",msg);
#endif
}

/* application-dependent functions  */

/* read the remaining time on timerTmCD and
   if valid, write it to register RemainingTime.
   The contents of this register will be displayed on the simulator */

void refreshTimerDisplay ()
{
  int k;
```

```
  if ( (k = RTLTimerCount(timerCD)) != -1 ) {
    RTL_putRemainingTime((Binary) (k/10)); /* divide by 10 for seconds */
  }
}

/* ---------------- MAIN ---------------- */


int main(int argc, char *argv[])
{
  int k;
  void (*presentState) (void);
  char buf[80];

  /* initializations */
  /* Clock: */
  /*       calibration */
  fprintf(stderr,"Clock calibration. Takes about 10 s.\n");
  clockCalibration();
  /*       granularity */
  setClockQuantum(100); /* 100 ms  <=> 10 Hz */

  /* Timers: */
        nbOfTimers = timerEnd;
        /* timerEnd is an application-dependent constant set in microwave.h */
        RTLTimerInit(); /* set up timer data structures */

  /* Connection to the simulator: */
    /*          establish connection */
    if (RTL_connect("localhost",2540)) exit(1);
    /*          connection data structures */
    if (RTL_init()) exit(2);

    RTL_get(); /* the server sends the registration number. Ignore it. */

        /* States: */
    initializeState(); /* set nextState to the initial state */

    /* query for initial actuator values */
    Actuator_set();
  presentState = nextState;

    /* ready to execute */
  (* nextState) (); /* do entry actions of the intial state */

    /* time-constrained execution starts from now */
  /* set the clock to 0 */
  updateClock();
  /* send initial actuator values to the simulator */
    RTL_put();

  /* you can now loop on operations */
    while(1) {
          while (!updateClock()) ; /* busy wait */
      /* starting a new control loop */
      /* update timers */
      RTLTimerCheck(); /* side effect: may generate event timeouts */
        /* get operands set in the Tcl-Tk window */
        RTL_get();
```

```
        /* Here all the simulator-set registers are updated
            (RTL_status and sens)  */
    /* Sensor_edit(stderr,'d'); */
    /* fprintf(stderr,"InEvent = %s\n",Binary2text(InEvent,'b',buf)); */
       setEvents(); /* capture incoming events */

        if (shouldIQuit()) break; /* exit the loop */

        /* Do the actual control */
    (*nextState)(); /* execute the reaction */
    if (presentState != nextState) {
      /* transition in progress. Do entry actions */
      (* nextState)();
      presentState = nextState;
    }
    refreshTimerDisplay();
        /* send the value of all controller-set registers to the simulator */
        RTL_put();
        /* reset of events */
        resetEvents();
    }
    /* exit this infinite loop by a break */
    /* Epilog: close connection */
    fprintf(stderr,"That's All Folks!\n");
    RTL_shutdown();

    return 0;
}
```

6 microwave.c

```
/* microwave.c */
/* Charles André */
/* January 4, 2000 */
/* December 17, 2000:  adaptation to the new distribution */

#define GLOBAL_MICROWAVE_
#include "microwave.h"
#include <stdio.h>

/* auxiliary functions */
extern void debugMsg(const char *);



/* functions associated with timers */

static void genevDone(void)
{
  present[evDone] = true;
  debugMsg("evDone is present");
}

static void genevTmDD(void)
{
  present[evTmDD] = true;
  debugMsg("evTmDD is present");
}
```

```
static void genevTmUD(void)
{
  present[evTmUD] = true;
  debugMsg("evTmUD is present");
}


static void computeDurations (void)
{
  cd = RTL_getCookingDuration()*10;/* 10 tu = 1s */
  ud = 2 * RTL_getPowerSelection(); /* recursion period = 2000 ms for pwm = 20
  dd = 20 - ud;
  /* set up of timers */
  RTLTimerSetPeriod(timerCD,0);/* aperiodic */
  RTLTimerSetPeriod(timerUD,0);
  RTLTimerSetPeriod(timerDD,0);
  RTLTimerSetFunc(timerCD,genevDone);
  RTLTimerSetFunc(timerUD,genevTmUD);
  RTLTimerSetFunc(timerDD,genevTmDD);
  /* Start timerCD (Cooking duration) */
  RTLTimerStart(timerCD,cd);
}


/* event management */
void resetEvents(void)
{
  Events e;
  for (e=evNone;e<evEnd;e++) present[e] = false;
}

void setEvents(void)
{
  if (RTL_getDoorOpens())        present[evDoorOpens] = true;
  if (RTL_getFlameON())          present[evFlameON] = true;
  if (RTL_getCookingSelected()) present[evCookingSelected] = true;
  if (RTL_getCancel())           present[evCancel] = true;
}

/* forward declaration of all the states, in order to avoid implicit
prototype */

static void State_Idle (void);
static void State_Wait (void);
static void State_POff (void);
static void State_POn (void);
static void State_PauseOff (void);
static void State_PauseOn (void);

/* State section */

void initializeState(void)
{
  resetEvents();
  nextState = State_Idle;
}

/* exit fuctions:
   They have been set apart for a better code structuration
```

```
   */

static exit_State_POff (void)
{
  RTL_putLight(OFF);
  RTL_putFan(OFF);
  RTL_putSpit(OFF);
  /* debugMsg(">>Leaving State_POff"); */
}

static exit_State_POn (void)
{
  RTL_putLight(OFF);
  RTL_putFan(OFF);
  RTL_putSpit(OFF);
  RTL_putPower(OFF);
  /* debugMsg(">>Leaving State_POff"); */
}

/* This state has no entry, no exit, no activity;
   a single transition without guard */
static void State_Idle (void)
{
  static int entering = true;

  /* entering */
  if (entering) {
    debugMsg(">> Entering State_Idle");
    entering = false;
    return;
  }

  /* steady state */

  /* check transitions for leaving */
  if (present[evCookingSelected]) {
    nextState = State_Wait;
  } else {
    /* stay in this state */
          /* nextState = State_Idle; */
    return;
  }

  /* leaving */
  entering = true; /* for future entering */
}

/* This state has no entry, no exit, no activity;
   2 transitions:
    one without guard,
    one with guard */
static void State_Wait (void)
{
  static int entering = true;

  /* entering */
  if (entering) {
    debugMsg(">> Entering State_Wait");
    entering = false;
```

```
      return;
    }

    /* steady state */

    /* check transitions for leaving */
    if (present[evCancel]) {
      nextState = State_Idle;
    } else {
      if (present[evFlameON] && RTL_getDoorClosed()) {
        nextState = State_POff;
        /* transition actions */
        computeDurations();
      } else {
        /* stay in this state */
              /* nextState = State_Wait; */
        return;
      }
    }

    /* leaving */
    entering = true; /* for future entering */
  }

  /* State with entry, exit actions, no activity;
    4 transitions without guard */
  static void State_POff (void)
  {
    static int entering = true;

    /* entering */
    if (entering) {
      debugMsg(">>Entering State_POff");
      entering = false;
      /* entry actions */
      RTL_putLight(ON);
      RTL_putFan(ON);
      RTL_putSpit(ON);
      RTL_putPower(OFF);
      /* start timerDD (duration of power down) */
      RTLTimerStart(timerDD,dd);
      return;
    }
    /* steady state */

    /* check transitions for leaving */
    if (present[evCancel]) {
      exit_State_POff();/* perform exit actions */
      RTLTimerStop(timerCD);
      RTLTimerStop(timerDD);
      nextState = State_Idle;
    } else {
      if (present[evDone]) {
        exit_State_POff(); /* perform exit actions */
        RTLTimerStop(timerCD);
        RTLTimerStop(timerDD);
        nextState = State_Idle;
      } else {
        if (present[evDoorOpens]) {
```

```
            exit_State_POff(); /* perform exit actions */
            nextState = State_PauseOff;
            /* transition actions */
            RTLTimerSuspend(timerDD);
            RTLTimerSuspend(timerCD);
          } else {
            if (present[evTmDD]) {
              exit_State_POff(); /* perform exit actions */
              nextState = State_POn;
            } else {
              /* stay in this state */
                    /* nextState = State_POff; */
              return;
            }
          }
        }
      }

      /* leaving */
      entering = true; /* set for future entering */
    }

    static void State_POn (void)
    {
      static int entering = true;

      /* entering */
      if (entering) {
        debugMsg(">>Entering State_POn");
        entering = false;
        /* entry actions */
        RTL_putLight(ON);
        RTL_putFan(ON);
        RTL_putSpit(ON);
        RTL_putPower(ON);
        /* start timerUD (duration of power up) */
        RTLTimerStart(timerUD,ud);
        return;
      }

      /* steady state */

      /* check transitions for leaving */
      if (present[evCancel]) {
        exit_State_POn();/* perform exit actions */
        RTLTimerStop(timerCD);
        RTLTimerStop(timerUD);
        nextState = State_Idle;
      } else {
        if (present[evDone]) {
           exit_State_POn(); /* perform exit actions */
           RTLTimerStop(timerCD);
           RTLTimerStop(timerUD);
          nextState = State_Idle;
        } else {
          if (present[evDoorOpens]) {
            exit_State_POn(); /* perform exit actions */
            nextState = State_PauseOn;
            /* transition actions */
```

```
          RTLTimerSuspend(timerUD);
          RTLTimerSuspend(timerCD);
        } else {
          if (present[evTmUD]) {
            exit_State_POn(); /* perform exit actions */
            nextState = State_POff;
          } else {
            /* stay in this state */
                  /* nextState = State_POn; */
            return;
          }
        }
      }
    }

    /* leaving */
    entering = true; /* set for future entering */
  }

  static void State_PauseOff (void)
  {
    static int entering = true;

    /* entering */
    if (entering) {
      debugMsg(">> Entering State_PauseOff");
      entering = false;
      return;
    }

    /* steady state */

    /* check transitions for leaving */
    if (present[evCancel]) {
      nextState = State_Idle;
      RTLTimerStop(timerCD);
      RTLTimerStop(timerDD);
    } else {
      if (RTL_getDoorClosed()) {
        nextState = State_POff;
        /* transition actions */
        RTLTimerResume(timerDD);
        RTLTimerResume(timerCD);
      } else {
        /* stay in this state */
        nextState = State_PauseOff;
        return;
      }
    }

    /* leaving */
    entering = true; /* set for future entering */
  }

  static void State_PauseOn (void)
  {
          static boolean entering = true;

    /* entering */
```

```
    if (entering) {
      debugMsg(">> Entering State_PauseOn");
      entering = false;
      return;
    }

    /* steady state */

    /* check transitions for leaving */
    if (present[evCancel]) {
      nextState = State_Idle;
      RTLTimerStop(timerCD);
      RTLTimerStop(timerUD);
    } else {
      if (RTL_getDoorClosed()) {
        nextState = State_POn;
        /* transition actions */
        RTLTimerResume(timerUD);
        RTLTimerResume(timerCD);
      } else {
        /* stay in this state */
        nextState = State_PauseOn;
        return;
      }
    }

    /* leaving */
    entering = true; /* set for future entering */
  }
```