RTSSim - A Simulation Framework for Complex Embedded Systems

Johan Kraft
School of Innovation, Design and Engineering
Mälardalen University, Västerås, Sweden
johan.kraft@mdh.se

April 9, 2009

Abstract

This report presents the current state of RTSSim, a simulation framework for complex embedded systems focusing on timing and resource usage properties. The report presents the core concepts of RTSSim, the elements of RTSSim simulation models and associated operations, as well as an example of a fairly complex RTSSim simulation model.

1 Overview

RTSSim has been developed to allow for simulation-based analysis of task timing, resource usage and other dynamic properties of embedded systems, which are hard to understand from the source code alone.

A simulation model for RTSSim is expressed in plain C code and is focused on tasks (i.e. processes/threads). Currently, only single CPU-core models are supported, but it is possible to model input from an environment (other systems or CPU cores). The tasks in the simulation model describes the tasks of a real or fictive embedded system, with a focus on behavior relevant for timing or resource usage properties, such as task interactions (communication, synchronization), task attribute changes (e.g. priority) and important state changes.

Each task in RTSSim is a C program, which executes in a "sandbox" environment with similar services and runtime mechanisms as a normal real-time operating system. The scheduling policy or RTSSim is preemptive fixed-priority scheduling and each task has scheduling attributes such as priority, periodicity and offset. It is possible to change these parameters dynamically, in the task model code, to implement a custom scheduling policy.

An RTSSim simulation is performed by compiling the model together with the RTSSim library (in a Microsoft Visual Studio project) and running the resulting executable. The output is, depending on configuration, either a detailed

trace file for the Tracealyzer tool [?], or a textfile with timing statistics for a specific task.

Compared to traditional testing, an RTSSim simulation is often 100-1000 times faster than running real test-cases on the target system. This is partly due to the typically higher abstraction-level of the simulation model and faster CPU of desktop computers. Moreover, RTSSim can run large amounts of simulations automatically and explore different scheduling/timing scenarios much more efferently than ordinary testing. The simulator can be used in two ways, depending on the properties of interest. For estimation of average case behavior (performance), Monte-Carlo (random) simulation is used. For estimation of extreme values, two methods for simulation optimization have been developed in our research, MABERA [?] and HCRR (publication pending). In these methods, RTSSim is used in an iterative manner by an optimization algorithm, which tries to find as extreme values as possible for the specified property, e.g., task response time. Note that these methods are still "best effort" approaches, just like Monte-Carlo simulation, but are significantly more efficient in finding extreme values for non-trivial simulation models.

2 Concepts of RTSSim

RTSSim models time using an integer variable, *clk*. All time-dependant operations of RTSSim (task activations, timeouts, delays and simulation termination) depends on clk. The clk variable is only advanced explicitly, through a special RTSSim library routine. Due to this "virtual time", the timing behavior of the host computer does not affect the simulation result.

An RTSSim model may contain three types of RTSSim-specific elements: tasks, message boxes and semaphores.

An RTSSim simulation model typically contains stochastic selections. The most visible is the jitter attribute of tasks, which adds a random variation in task release time, enabling modeling of stochastic tasks. RTSSim determines stochastic selections either using pseudo-random numbers generated during runtime (resulting in random monte-carlo simulation), or from values specified as parameters to RTSSim when it is controlled by a simulation optimization algorithm. When using random simulation, RTSSim reports the "seed value" used to initiate the random number generator. Thereby it is possible to replicate previous simulations. Another types of stochastic selections are execution time variations (stochastic increment of the simulation clock) and model-specific stochastic selections, typically in tasks used to model the environment.

Note that RTSSim is completely deterministic when used in simulation optimization approaches such as HCRR or MABERA. The term "stochastic selections" is a bit misleading when RTSSim is used in that context, as such selections are directly controlled by the input data to RTSSim in that case. However, the "stochastic selections" are in some sense always stochastic from a model point-of-view, as the model does not decide their outcome.

3 Tasks

The tasks define the behavior of the simulation. A task has a name, a set of scheduling attributes (priority, periodicity, offset and jitter) as well an entry function. The entry function (which is automatically called by the RTSSim scheduler) is a C function taking one parameter, a pointer to its task control block, which it needs for using the RTSSim library routines. Tasks may be periodic, sporadic or "one-shot", depending on the attributes period and jitter. A period of -1 implies a one-shot task, i.e., a task that is only activated once (at the offset time). Sporadic tasks are specified by a positive period (minimum interarrival time) and a positive, non-zero jitter (the maximum variation in interarrival time), meaning that it has a variable but bounded interarrival time. Periodic tasks are specified using a positive period and a jitter of 0. The entry function of periodic and sporadic tasks should return to RTSSim and may therefore not contain any infinite main loops; this is however allowed for one-shot tasks.

3.1 Creating a task using createTask

TCB createTask(char* name, int prio, int period, int offset, int jitter, void (*func)(void* TCB))*

This routine creates and initiates a task in RTSSim and is normally called from the "model_init" function, but may also be used dynamically, from the task code. A task has four scheduling attributes:

- prio: The scheduling priority (importance) of the task. Lower values are more significant, so 0 is the top priority.
- period: The periodicity of the task. If -1, the task is a one-shot task, meaning that it is only released once (at current time (clk) + offset)
- offset: Allows separation of tasks in time. If the offset of a task is 100 and period is 4000, the task will be released at 100, 4100, 8100 etc.
- jitter: Specifies the maximum jitter of sporadic tasks. If the jitter of a task is 100 and period is 4000, the first instance of the task will be released somewhere in the interval [0..99], the second activation will occur [4000..4099] time units *after the previous release of the task*. Thus, the average interarrival time of this task is around 4050.

3.2 Advancing the simulation clock using execute

void execute(int duration)

The execute routine is used to advance the simulation clock (clk), which drives the simulation forwards. The size of the clock advance is specified in the duration parameter and should model the real modeled system's CPU usage for a

particular pieces of code. This information is typically obtained from detailed measurements, or from estimations if the modeled system is not implemented. Depending on the model's level of abstraction, a call to the execute routine may represent the code of a whole task or a smaller section of code. During the duration of the execute call, the task may be preempted by other tasks. In that case, RTSSim remembers the amount of execution time left to consume, and continues consuming the remaining CPU time when the task is again allowed to execute. Note that preemptions does not occur outside RTSSim library functions, so "normal" C code is executed in a non-preemtive manner. To allow preemptions in between such C statements (other statements than calls to RTSSim library routines), an execute statement should be added in between. Random execution time variations can be implemented by passing pseudo-random values (in a suitable range) to the execute routine.

3.3 Waiting for a specified duration using delay

void delay(int duration)

A call to the delay routine puts the task to sleep for *duration* time units. After the specified time has passed, the calling task becomes ready for execution.

4 Message boxes

A message box is a FIFO buffer storing messages between tasks. An RTSSim task may put messages in the message box using the sendMessage library routine and fetch messages using the recvMessage library routine. A message is a 32-bit positive integer value, which typically represents a message code (a service request or status answer). Negative values are not recommended, as the recvMessage operation uses negative return values to represent error codes (e.g. timeout).

4.1 Creating a message box using createMBOX

MBOX createMBOX(char name, int size);*

Creates a message box with specified name and size, i.e. the maximum number of messages the messagebox can store before blocking occurs. Returns a handle necessary for later sendMessage and recvMessage calls on the created message box.

4.2 Sending a message using sendMessage

int sendMessage(MBOX mbox, int msg, int timeout)

Attempts to send message msg is to the messagebox mbox. If mbox is full,

the sending task is blocked until there is room for the message, or a timeout occurs. If timeout parameter is FOREVER, no timeout will occur (it waits "forever"). If the timeout is 0, a timeout occurs immediately if there is no empty slot in the messagebox. If a timeout occurs, the return code is TIMEOUT, otherwise OK. The message should be a single 32-bit integer value. Negative values are not allowed, as recvMessage uses negative values for error return codes.

Example:

```
#define MY_MSG_CODE 123
int status;
status = sendMessage(MyMBOX, MY_MSG_CODE, 1000);
if (status == TIMEOUT)
{
    /* timeout error handling */
    ...
}
```

4.3 Receiving a message using recvMessage

int recvMessage(MBOX mbox, int timeout)

A message is received from the messagebox mbox. If mbox is empty, the task is blocked until a message arrives or the timeout occurs. If timeout is specified as FOREVER (-1), no timeout will occur. If timeout is specified to 0, the task is not blocked by an empty mbox, but immediately timeouts if no message is available.

Example:

```
int msg;
msg = recvMessage(MyMBOX, FOREVER);

switch (msg)
{
    case TIMEOUT:  /* timeout error handling */
                   break;
    case REQUEST1: ...
                   break;
    case REQUEST2: ...
                   break;
}
```

5 Semaphore

RTSSim provides a classic Djikstra binary semaphore, for mutual exclusion between tasks. A semaphore is locked using the sem_wait library routine (corre-

sponding to Djikstra's P) and released using sem_post routine (corresponding to Djikstra's V).

5.1 Creating a Semaphore using createSemaphore

SEMAPHORE createSemaphore(char name);*

Creates a semaphore with specified name. To lock the new semaphore, a call to sem_wait is also required. Returns a handle necessary for later sem_wait and sem_post calls on the created semaphore.

5.2 Locking a Semaphore using sem_wait

int sem_wait(SEMAPHORE sem, int timeout)

Attempts to lock a semaphore. If the semaphore is already locked (typically by another task), the operation blocks until it is allowed to lock the semaphore or the timeout occur. If the semaphore was locked, the return code is OK, otherwise, if a timeout occurs, the return code is TIMEOUT. The time duration to wait before timeout is specified in the timeout parameter. If this is 0, a timeout will immediately occur if the semaphore was already locked. If the timeout is specified to FOREVER, no timeout will occur.

Example:

```
int status;
status = sem_wait(MySem, 10000);

if (status == TIMEOUT)
{
    /* failed locking the semaphore */
    ...
}
```

5.3 Releasing a Semaphore using sem_post

int sem_post(SEMAPHORE sem)

A previously locked semaphore is unlocked. If other tasks are waiting to lock the semaphore, they will be made ready to execute.

Example:

```
sem_post(MySem);
```

6 Modeling the system environment

As previously mentioned, RTSSim can only simulate the scheduling on one CPU, but is is possible to model the input from other nodes, I/O interfaces, sensors etc, by using *environment tasks*. Such tasks should not call the execute routine and are therefore "invisible" in the simulation, except for their intended impact, as they don´t consume any CPU time. Such tasks should have top-priority, i.e., priority 0.

7 Simulation output

RTSSim can generate two types of output, either a detailed trace of the simulation, or a text file with statistics for a specific task. The former is used to study the details of a simulation and the latter when using RTSSim in a simulation optimization algorithm, such as MABERA or HCRR. The statistic text file contains the highest response time and execution time for a specific task. If using RTSSim in batch mode, i.e., in order to run several independent simulations, the textfile will contain the result of each simulation result. The trace output option generates a binary file for the graphical Tracealyzer tool [**?**], containing an exact, timed trace of the task scheduling, communication events and model-specific events. An example of the trace view is depicted by Figure **??**.

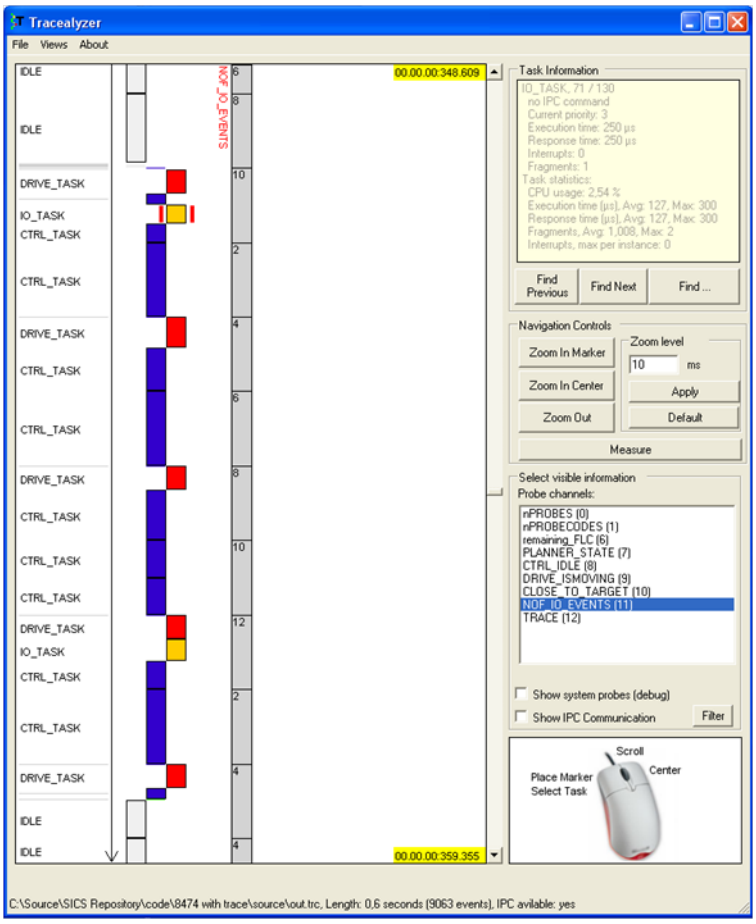

Figure 1: A Tracealyzer view of the example RTSSim model presented in Section ?? (specifically the extreme scenario described in Section ??)

8 Example model

This section gives an example of the essential parts of a fairly complex RTSSim model. This model was used for the evaluation of the MABERA method for simulation optimization, presented in [?]. This model describes a fictive system, but is inspired a control system for industrial robots, developed by ABB. The tasks of this model violate several assumptions of the traditional methods for analytical response-time analysis. The tasks in the model may:

- trigger the execution of other tasks through communication using message queues,
- be triggered both by timers and events, or a combination of both,
- have different temporal behaviors depending on the contents of received messages and the value of shared state variables,
- be blocked on sending and receiving of messages, and
- change the scheduling priority of tasks as a response to certain events.

The modeled fictive system controls a set of electric motors based on periodic sensor readings and aperiodic events. The calculations necessary for a real control system is not included in this model, the model mainly describes execution time, communication and other behavior that impact the temporal behavior. The model contains four periodic tasks:

Task	Priority	Period
PLAN_TASK	50	40000 or 10000
CTRL_TASK	40 or 20	10000 or 20000
IO_TASK	30	5000
DRIVE_TASK	10	2000

An overview of the model is given in Figure ??, where colors are used to indicate priority (red indicates top priority, yellow medium priority and green lowest priority. The illustration also shows the message queues (named XXQ) which the tasks use to communicate. The queue DDQ is critical in the application and not allowed to become empty.

PLAN_TASK is responsible for high level planning of how to move the physical object connected to the motors. It periodically sends coordinates to CTRL_TASK through the queue CDQ (CTRL Data Queue). CTRL_TASK calculates control references for the motors with respect to input from CDQ and from IO_TASK, through the queue IOQ. The resulting motor control references from CTRL_TASK are sent to DRIVE_TASK, through DDQ, which controls the motors. The purpose of IO_TASK is to collects buffered I/O events from the system's environment (and send this information to CTRL_TASK). Depending on the physical state of the controlled system, different numbers of I/O messages

are received from the environment (e.g., sensors). The number of incoming messages for IO_TASK are modeled using the integer variable *nofEvents*, which is increased by the environment task IO_ENVTASK, by 0, 1 or 2, every 1000 time units. IO_TASK, which has a period of 5000, decreases this variable by 1 for each message that is sent to IOQ. The increments of nofEvents in IO_ENVTASK is a simulator input (or random if monte-carlo simulation).

As indicated by the table, both CTRL_TASK and PLAN_TASK may change priority and periodicity in response to specific events in the model. The period of CTRL_TASK is normally 20000 time units, but when a movement is approaching the target, the period is decreased to 10000 in order to improve control performance. The priority of CTRL_TASK is boosted if the input queue for DRIVE_TASK has decreased below a certain threshold, since this queue must never become empty. PLAN_TASK uses a shorter periodicity when idle, in order to faster detect a start event.

There are three types of events from the system´s environment: START, STOP and GETSTATUS. These events are sent to PLAN_TASK through the queue PCQ (PLAN Command Queue), which processes them accordingly; some are forwarded to CTRL_TASK and DRIVE_TASK, through their command queues CCQ and DCQ. The START event will cause the system to change state into active, which means that it powers up and controls the motors. The STOP event causes the system to power down the motors and go to idle state. The GETSTATUS event causes all tasks to send a status message to the user interface (an environment task). These events impact the execution time of the tasks. The events are generated by the environment tasks GETSTATUS_ENVTASK, START_ENVTASK and STOP_ENVTASK.

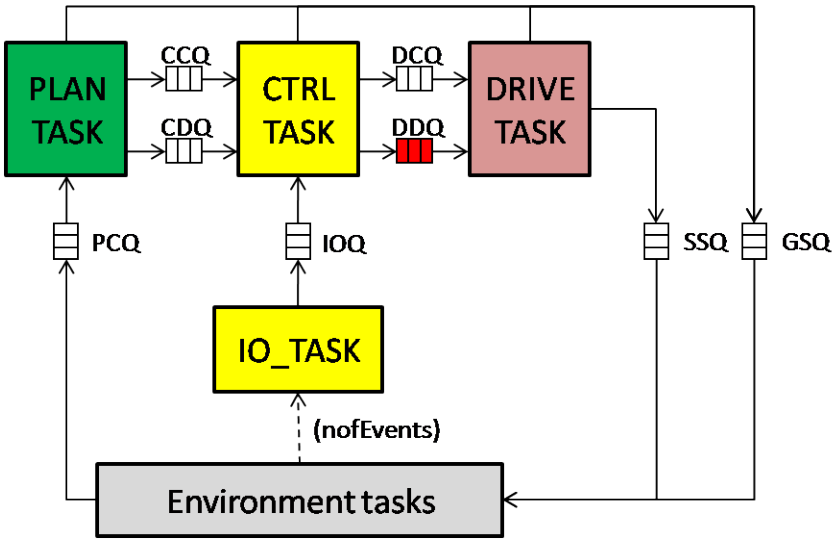

Figure 2: The tasks and communication dependencies of the example model

8.1 An extreme scenario

This model can not be analyzed using traditional methods such as RTA, but using a simulation optimization method, HCRR, we have identified an extreme scenario regarding the response time of CTRL_TASK, the most complex task in the model, which is believed to be the worst case response time. The response time of CTRL_TASK is in the average case around 3200, but can reach a value of 8474. The scenario, which is depicted by in the example trace view of Figure ??, depends on the following conditions:

- The number of messages in IOQ is 32 when the critical instance of CTRL_TASK begins to execute. This is very high, in fact the largest IOQ size observed in any experiment on this model.
- An instance of IO_TASK preempts the critical CTRL_TASK instance and refills IOQ with 10 messages during the CTRL_TASK's IOQ read loop, increasing the iterations of this loop from 32 to 42.
- A rare sporadic event (GETSTATUS) had just occurred, which results in messages for the following instance of CTRL_TASK and DRIVE_TASK, which increase their execution times.
- As a result of the long execution time of the critical instance of CTRL_TASK (6224), it is preempted by five instances of DRIVE_TASK (one with unusually long execution time, due to a preceding GETSTATUS event) and two instances of IO_TASK.

The number of messages in IOQ has a major impact on the execution time of CTRL_TASK. The number of messages in IOQ is increased when IO_TASK executes, every 5000 time units, and depends on the global variable nofEvents. Maximum 12 messages are sent to IOQ at each instance of IO_TASK. The nofEvents variable is in turn increased by an environment task, IO_ENVTASK, which executes every 1000 time units. This increases nofEvent by 0, 1 or 2 (according to simulator input data or random selection, depending on mode). Reaching an IOQ size of 32 required an intricate sequence of input data (i.e. the nofEvent increase by 0, 1 or 2); if always increasing by 2, the IOQ size becomes maximum 30, resulting in a response time of 8324, compared to the worst case of 8474. The reason for this is in the relative timing between previous instances of CTRL_TASK and IO_TASK:

In the worst-case scenario identified, the CTRL_TASK instance preceding the critical instance had only 3 messages in IOQ to consume, which allowed it to finish the read loop before IO_TASK refilled it, which implied that these messages were instead processed by the next (the critical) CTRL_TASK instance.

In the "tweaked" case, where only 2:s where given as input (i.e., added to nofEvents), the previous CTRL_TASK instance had more messages in IOQ to consume, which took longer time and made the IO_TASK preempt and refill IOQ during the read-loop. Thereby, these messages were consumed by this

previous task instance, which decreased the IOQ size for the next (the critical) instance of CTRL_TASK.

The large IOQ size in this case (32) was partly caused by DRIVE_TASK; it increased the priority of CTRL_TASK momentarily, as the number of messages in DDQ has dropped below a specified threshold. This is a mechanism to prevent buffer-underrun situations on DDQ (it may not become empty) and implies that instances of IO_TASK are delayed, which changes the relative timing between IOQ's producer (IO_TASK) and consumer (CTRL_TASK).

8.2 Model code

```
void PLAN_TASK(TCB* tcb)
{
   int nFLCs;
   int cmd;
   SUBSYSTEM_DATA* subsystem = tcb->userdata;

   // process all pending requests in PCQ
   do
   {
      cmd = recvMessage(tcb, subsystem->PCQ, 0);
      execute(tcb, cPLANdecode);
      if (cmd != -1)
      {
         switch(cmd)
         {
            case MSG_START:
            subsystem->remainingFLC = 130;
            trcrec_store_probe(subsystem->probe_remaining_FLC,
                               subsystem->remainingFLC);
            subsystem->planstate = PLANSTATE_BEGIN;
            trcrec_store_probe(subsystem->probe_plan_task_state,
                               subsystem->planstate);
            execute(tcb, cPLANstart);
            break;

            case MSG_STOP:
            subsystem->planstate = PLANSTATE_IDLE;
            trcrec_store_probe(subsystem->probe_plan_task_state,
                               subsystem->planstate);
            execute(tcb, cPLANstop);
            break;

            case MSG_GETSTS:
            execute(tcb, cPLANgetsts);
            sendMessage(tcb, subsystem->GSQ, MSG_STS_PLAN, FOREVER);
            sendMessage(tcb, subsystem->CCQ, MSG_GETSTS, FOREVER);
            break;

            default:
            sim_fail_int("PLAN_TASK got message: %d\n", cmd);
         }
      }
   }while (cmd != -1); // until no more messages
```

```
// execute periodic behavior, depending on state
switch (subsystem->planstate)
{
   case PLANSTATE_BEGIN:
   subsystem->planstate = PLANSTATE_WORKING;
   trcrec_store_probe(subsystem->probe_plan_task_state,
                      subsystem->planstate);
   subsystem->closeToTarget = 0;
   trcrec_store_probe(subsystem->probe_close_to_target,
                      subsystem->closeToTarget);

   if (subsystem->remainingFLC < CDQSIZE)
   {
      nFLCs = subsystem->remainingFLC;
   }else{
      nFLCs = CDQSIZE;
   }
   while (nFLCs > 0)
   {
      execute(tcb, cPLANflc);
      sendMessage(tcb, subsystem->CDQ, MSG_FLC, FOREVER);
      nFLCs--;
      subsystem->remainingFLC--;
   }
   tcb->period = 40000;
   break;

   case PLANSTATE_WORKING:
   if (subsystem->remainingFLC < 4)
   {
       nFLCs = subsystem->remainingFLC;
   }else{
       nFLCs = 4;
   }

   while (nFLCs > 0)
   {
       execute(tcb, cPLANflc);
       sendMessage(tcb, subsystem->CDQ, MSG_FLC, FOREVER);
       nFLCs--;
       subsystem->remainingFLC--;
   }
   tcb->period = 40000;
   break;

   case PLANSTATE_IDLE:
```

```
      tcb->period = 10000;
      break;
   }

   trcrec_store_probe(subsystem->probe_remaining_FLC,
                      subsystem->remainingFLC);

   if (((subsystem->remainingFLC <= 0) &&
       (subsystem->planstate != PLANSTATE_IDLE)) ||
       ((subsystem->remainingFLC > 0) &&
       (subsystem->planstate == PLANSTATE_IDLE)))
   {
      execute(tcb, cPlanLast);
      subsystem->planstate = PLANSTATE_IDLE;
      subsystem->closeToTarget = 1;
      trcrec_store_probe(subsystem->probe_close_to_target,
                         subsystem->closeToTarget);
      subsystem->remainingFLC = 0;
      sendMessage(tcb, subsystem->CDQ, MSG_LAST, FOREVER);
      trcrec_store_probe(subsystem->probe_plan_task_state,
                         subsystem->planstate);
   }
}

void CTRL_TASK(TCB* tcb)
{
   int msg = -1;
   int ioevent;
   int i;
   int nSLC = -1;
   int nofIOEvents = 0;
   SUBSYSTEM_DATA* subsystem = tcb->userdata;

   msg = recvMessage(tcb, subsystem->CCQ, 0);
   execute( tcb, cCTRLdecode );

   if (msg > -1)
   {
      switch (msg)
      {
         case MSG_GETSTS:
         sendMessage(tcb, subsystem->GSQ, MSG_STS_CTRL, FOREVER);
         execute(tcb,cCTRLgetsts);
         sendMessage(tcb, subsystem->DCQ, MSG_GETSTS, FOREVER);
         break;
```

```
      default:
      sim_fail_int("CTRL_TASK got message: %d\n", msg);
      break;
   }
}

// consume all IO events
i = 0;
do{
   ioevent = recvMessage(tcb, subsystem->IOQ, 0);
   if (ioevent > -1)
   {
      i++;
      execute(tcb, cCTRLioevent);
   }
}while (ioevent > -1);

if (subsystem->closeToTarget == 0)
{
   nSLC = 10;
   tcb->period = 20000;
}
else
{
   nSLC = 5;
   tcb->period = 10000;
}

// Process any FLC message from PLAN_TASK
msg = recvMessage(tcb, subsystem->CDQ, 0);
if (msg > -1)
{
   switch(msg)
   {
      case MSG_FLC:
      if (subsystem->idle == 1)
      {
         subsystem->idle = 0;
         trcrec_store_probe(subsystem->probe_ctrl_idle,
                            subsystem->idle);
      }
      while (nSLC-- > 0)
      {
         // generate SLC data to DRIVE
         execute(tcb,cCTRLslc);
         sendMessage(tcb, subsystem->DDQ, MSG_SLC, FOREVER);
```

```
         }
         break;

         case MSG_LAST:
         subsystem->idle = 1;
         subsystem->closeToTarget = 0;
         execute(tcb,cCTRLlast);
         trcrec_store_probe(subsystem->probe_ctrl_idle,
                            subsystem->idle);
         break;

         default:
         sim_fail_int("CTRL_TASK got message %d\n", msg);
         break;
      }
   }
   else // if no message
   {
      if (subsystem->idle == 0)
      {
         // if expecting message
         sim_fail("CTRL_TASK starvation!\n");
      }
   }

   // if idle, generate default data (slcd)
   if (subsystem->idle == 1)
   {
      while (nSLC-- > 0)
      {
         execute(tcb,cCTRLslcd);
         sendMessage(tcb, subsystem->DDQ, MSG_SLCD, FOREVER);
      }
   }
}

void DRIVE_TASK(TCB* tcb)
{
   int msg;
   SUBSYSTEM_DATA* subsystem = tcb->userdata;

   msg = recvMessage(tcb,subsystem->DDQ,0);

   execute(tcb,cDRIVEdecode);

   if (msg == -1)
```

```
{
   sim_fail("DRIVE_TASK starvation!\n");
}

if (subsystem->DDQ->current_size < MINDDQSIZE)
{
   // boost priority of CTRL_TASK, now higher than IO_TASK
   subsystem->ctrl_task->prio =
                          20 + subsystem->subsystem_index;
}
else
{
   // normal priority of CTRL_TASK, lower than IO_TASK
   subsystem->ctrl_task->prio =
                          40 + subsystem->subsystem_index;
}
trcrec_store_probe(subsystem->probe_ctrl_prio,
                   subsystem->ctrl_task->prio);

// process data message from CTRL_TASK
switch(msg)
{
   case MSG_SLC:
   execute(tcb, cDRIVEslc);
   if (subsystem->ismoving == 0)
   {
      subsystem->ismoving = 1;
      trcrec_store_probe(subsystem->probe_drive_ismoving,
                         subsystem->ismoving);
      sendMessage(tcb, subsystem->SSQ, MSG_MOVING, FOREVER);
   }
   break;

   case MSG_SLCD:
   execute(tcb, cDRIVEslcd);
   if (subsystem->ismoving == 1)
   {
      subsystem->ismoving = 0;
      trcrec_store_probe(subsystem->probe_drive_ismoving,
                         subsystem->ismoving);
      sendMessage(tcb, subsystem->SSQ, MSG_NOTMOVING, FOREVER);
   }
   break;

   default:
   sim_fail_int("Warning, DRIVE_TASK unhandled message: %d\n", msg);
```

```
      break;
   }
   // check for command, i.e., any getstatus requests
   msg = recvMessage(tcb, subsystem->DCQ, 0);
   if (msg > -1)
   {
      switch(msg)
      {
         case MSG_GETSTS:
         execute(tcb, cDRIVEgetsts);
         sendMessage(tcb, subsystem->GSQ, MSG_STS_DRIVE, FOREVER);
         break;

         default:
         sim_fail_int("DRIVE_TASK got message %d\n", msg);
         break;
      }
   }
}

void IO_TASK(TCB* tcb)
{
   int status;
   int eventsToProcess = 0;
   SUBSYSTEM_DATA* subsystem = tcb->userdata;

   if (subsystem->nofEvents > 12)
   {
      // save some IO events for next job
      eventsToProcess = 12;
   }
   else
   {
      // normal case, process all IO events
      eventsToProcess = subsystem->nofEvents;
   }

   while(eventsToProcess-- > 0)
   {
      execute(tcb, cIOEvent);
      subsystem->nofEvents--;
      status = sendMessage(tcb, subsystem->IOQ, 1, 0);
      if (status < 0)
      {
         printf("IOQ overflow! clk: %d\n", clk);
      }
```

```
   }
}


void IO_ENVTASK(TCB* tcb)
{
    nofEvents += (int)(getRandomValue() % 3);
}

void GETSTATUS_ENVTASK(TCB* tcb)
{
    int reply;

    sendMessage(tcb, PCQ, MSG_GETSTS, FOREVER);

    reply = recvMessage(tcb, GSQ, FOREVER);

    if (reply != MSG_STS_PLAN)
    {
        printf("Warning, got unexpected message %d\n",reply);
    }

    reply = recvMessage(tcb, GSQ, FOREVER);

    if (reply != MSG_STS_CTRL)
    {
        printf("Warning, got unexpected message %d\n",reply);
    }

    reply = recvMessage(tcb, GSQ, FOREVER);

    if (reply < 0)
        return;

    if (reply != MSG_STS_DRIVE)
    {
        printf("Warning, got unexpected message %d\n",reply);
    }
}

void START_ENVTASK(TCB* tcb)
{
    int reply;

    sendMessage(tcb,PCQ,MSG_START, FOREVER);
```

```
    reply = recvMessage(tcb, SSQ, FOREVER);

    if (reply != MSG_MOVING)
    {
        printf("Warning, got unexpected message %d\n",reply);
    }

    // create the STOP task dynamically, as a one-shot task (period -1)
    createTask("STOP_ENVTASK",
                0, // priority
                -1, // period (one-shot)
                clk + 100000, // offset (start time) is set to current time + 100000
                100000, // jitter
                STOP_ENVTASK);
}

void STOP_ENVTASK(TCB* tcb)
{
    int reply;

    sendMessage(tcb,PCQ,MSG_STOP, FOREVER);

    reply = recvMessage(tcb, SSQ, FOREVER);

    if (reply != MSG_NOTMOVING)
    {
        printf("Warning, got unexpected message %d\n",reply);
    }
}
```

References

[1] Tracealyzer website, http://www.tracealyzer.se.

[2] J. Kraft, Y. Lu, C. Norström, and A. Wall. A metaheuristic approach for best effort timing analysis targeting complex legacy real-time systems. In *Proc. of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 08)*, April 2008.