ISSN 0280–5316
ISRN LUTFD2/TFRT--7608--SE

TinyRealTime—An EDF Kernel for the Atmel ATmega8L AVR

Dan Henriksson
Anton Cervin

Department of Automatic Control
Lund Institute of Technology
February 2004

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 Lund Sweden

Document name
INTERNAL REPORT

Date of issue
February 2004

Document Number
ISRN LUTFD2/TFRT--7608--SE

Author(s)
Dan Henriksson, Anton Cervin

Supervisor

Sponsoring organisation

Title and subtitle
TinyRealTime—An EDF Kernel for the Atmel ATmega8L AVR

Abstract

This report describes the design and implementation of TinyRealTime, an event-based real-time kernel for the Atmel AVR ATmega8L 8-bit micro-controller. The kernel is event-based and supports fully preemptive earliest-deadline-first scheduling of tasks. Semaphores are provided to support task synchronization. The focus of the report is on the memory management, timing, and internal workings of the kernel.

The flash memory footprint of the kernel is approximately 1200 bytes and it occupies 11 bytes of SRAM memory for the kernel data structure plus an additional 11 bytes for each task and one byte for each semaphore. An application example is described, where the real-time kernel is used to implement concurrent control of two ball and beam laboratory processes using six application tasks.

Key words
Real-time kernel, Atmel AVR, Event-based, Earliest-deadline-first, Synchronization, Memory management, Control system application.

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
30

Recipient's notes

Security classification

Contents

1. Introduction

This report describes the design and implementation of TinyRealTime, a real-time kernel for the Atmel AVR ATmega8L 8-bit micro-controller. The kernel is event-based and supports fully preemptive earliest-deadline-first scheduling of tasks. Semaphores are provided to support task synchronization. The flash memory footprint of the kernel is approximately 1200 bytes and it occupies 11 bytes of SRAM memory for the kernel data structure plus an additional 11 bytes for each task and one byte for each semaphore.

To test the kernel, it was used to implement a control system for the ball and beam laboratory process. Two ball and beam processes were controlled concurrently using six application tasks. Each controller was implemented using a cascaded structure with one task for each loop in the cascade. Two additional tasks were used to implement a simple pulse width modulation of the output signal.

2. The Atmel AVR 8-bit RISC Processor

Below follows a summary of the features of the Atmel AVR RISC architecture and the ATmega8L micro-controller that were relevant in the implementation of the real-time kernel. The AVR has:

- 32 8-bit working registers,
- 8K bytes of in-system programmable flash memory,
- 1024 bytes of internal SRAM memory,
- throughput of ≈ 1 MIPS per MHz,
- 14.7456 MHz clock frequency,
- one 16-bit and two 8-bit timers/counters with separate pre-scalers, and compare match modes,
- four 10-bit resolution analog input channels,
- two analog output channels giving either $+10$ or -10 volt.

For a more detailed description on the Atmel AVR 8-bit RISC architecture, see [Atmel, 2003].

3. Kernel Implementation

The kernel implementation was divided into the solution of three main problems. The first problem was the memory management, since the AVR only has 1024 bytes of internal SRAM. The second area was timing and how to obtain event-based execution of the kernel. This area also included an interesting trade-off between the clock resolution and the system life-time. The last area was the internal workings of the kernel, and included issues such as context switching, and the representation of the ready queue and time queue of the kernel.

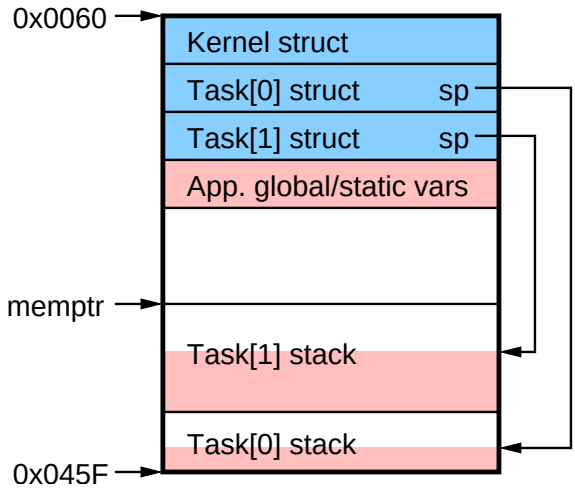

Figure 1 Memory layout of the real-time kernel.

3.1 Memory Layout and Data Structures

The AVR has 1120 memory locations of which the first 96 are used for the register file and the I/O memory, and the following 1024 (addresses $0x0060$ to $0x045F$) represent the internal SRAM.

In the kernel implementation, the 1024 bytes of SRAM are utilized according to the memory layout in Figure 1, showing the location of the kernel and task data structures and the individual stack memories. The data structures of the kernel are given by Listings 1 and 2.

As seen in Figure 1, the kernel data structure and the task data structures are allocated staticly from low addresses upwards followed by possible global and static variables for the particular application. Each task has an associated stack, and the stacks are allocated from the maximum address downwards. The stack sizes are specified by the user upon task creation, and it is the responsibility of the user not to exhaust the available memory.

A task occupies 11 bytes of memory, where 2 bytes are used to store the stack pointer of the task, 4 bytes each to represent the release time and absolute deadline of the task, and one byte to represent the state of the task. The kernel data structure occupies a total of 11 bytes of memory to represent; the number of tasks in the system, the currently running task, pointers to task and semaphore vectors, pointer to next available stack memory address (see Figure 1), number of major timer cycles (see Section 3.2), and the next wake-up time of the kernel.

Listing 1 The task data structure.

```
struct task {
  uint16_t sp;        // stack pointer
  uint32_t release;   // current/next release time
  uint32_t deadline;  // absolute deadline
  uint8_t state;      // terminated=0, readyQ=1, timeQ=2, semQ[]=3..
};
```

Listing 2 The kernel data structure. No dynamic memory allocation is allowed, so the task and semaphore vectors are allocated staticly depending on the user-specified constants `MAXNBRTASKS` and `MAXNBRSEMAPHORES`.

```
#define MAXNBRTASKS 6
#define MAXNBRSEMAPHORES 6

struct kernel {
  uint8_t nbrOfTasks;
  uint8_t running;
  struct task tasks[MAXNBRTASKS+1]; // +1 for the idle task
  uint8_t semaphores[MAXNBRSEMAPHORES];
  uint8_t *memptr;   // pointer to free memory
  uint16_t cycles;   // number of major cycles
  uint32_t nextHit;  // next kernel wake-up time
};
```

In order to reduce the RAM memory requirement of the kernel, no queues or sorted list functionality is implemented for the time queue, ready queue, and semaphore waiting queues. Instead, each task has an associated state, and linear search is performed in the task vector each time a task should be moved from the time queue to the ready queue, etc. The state will be any of: terminated (`state = 0`), ready (`state = 1`), sleeping (`state = 2`), or waiting on semaphore i (`state = 2+i`).

Depending of the maximum number of tasks and semaphores allowed (as specified by the user in the constants `MAXNBRTASKS` and `MAXNBRSEMAPHORES`) the total memory requirement of the kernel becomes

$$11 + 11 \cdot \text{MAXNBRTASKS} + \text{MAXNBRSEMAPHORES}$$

3.2 Timing

The output compare match mode of the 16-bit Timer/Counter 1 of the AVR is used to generate clock interrupts. Each time the timer value matches the compare match value, an interrupt is generated. The associated interrupt handler then contains the main functionality of the real-time kernel, such as releasing tasks, determining which ready task to run, and to perform context switches. This is detailed in Section 3.3.

Each time the kernel has executed, i.e., at the end of the output compare match interrupt routine, the output compare value is updated to generate a new interrupt at the next time the kernel needs to run. If this next wake-up time is located in a later major cycle (each timer cycle corresponds to 2^{16} timer ticks), the output compare value is set to zero. This way we make sure to get an interrupt at timer overflow to increase the `cycles` variable of the kernel data structure.

The timer uses a 16-bit representation and, as seen in the kernel data structure in Listing 2, an additional 16 clock bits are used to store major cycles. The time associated with each timer tick depends on the chosen pre-scaler factor of the timer (1, 8, 64, 256, or 1024).

Prescaler	*Clock resolution*	*Life time*
1	68 ns	5 min
8	543 ns	39 min
64	4.3 μs	5 h
256	17.4 μs	21 h
1024	69.4 μs	83 h

Table 1 Trade-off between clock resolution and system life time.

The choice of pre-scaler factor of the timer determines both the clock resolution and the system life time. No cyclic time is implemented, and thus the system life time is limited by the time it takes to fill all 32 clock bits in the time representation. The higher clock resolution (i.e., the smaller time between each timer tick), the shorter time before all 32 clock bits are filled. The life time and resolution for the different pre-scaler factors of the AVR are shown in Table 1.

The problem with limited life time versus high timing resolution can be avoided by using an implementation that uses a circular clock [Carlini and Buttazzo, 2003]. The extra cost introduced by this approach is that the clock needs to be checked for overrun at each invocation of the kernel, whereas an added advantage is that a fewer number of bits can be used for the clock representation (thus giving less computational overhead in every timing operation).

3.3 Kernel Internal Workings

As mentioned above, the main functionality of the kernel is implemented in the interrupt handler associated with the output compare match interrupt of the timer. The code that is executed in this handler is given in Listing 3 in Appendix A.

When the interrupt handler is entered, the stack pointer is having the address of the stack associated with the currently running task. The first step is then to store the status register and the 32 working registers on the stack. The code for this is automatically generated by the compiler. Next it is checked if there has been an overflow of the timer counter, in which case the number of major cycles is increased.

Thereafter, the task vector is gone through in order to determine if any tasks should be released at the current time, and to compute the closest release time of the remaining sleeping tasks. The ready task with the closest absolute deadline is then made the running task, and if this is not the same task that was executing at the start of the interrupt handler a context switch is performed. The context switch is done by saving the current address of the stack pointer in the task struct associated with the preempted task, and to update the stack pointer with the corresponding value of the new running task.

Finally, a new clock interrupt is set up, by updating the output compare match register. If the next clock interrupt will take place in a later major cycle, the output compare register is given the value zero. As seen in the code listing, some special care needed to be taken to not miss the overflow in the case that the current time is close to the beginning of the next major cycle.

Command	Description
`trtInitKernel`	Initialize the kernel.
`trtCreateTask`	Create a task.
`trtTerminate`	Terminate the execution of the current task.
`trtCurrentTime`	Get the current global time.
`trtSleepUntil`	Put a task to sleep until a certain time.
`trtGetRelease`	Retrieve the release time of the running task.
`trtGetDeadline`	Retrieve the absolute deadline of the running task.
`trtCreateSemaphore`	Create a semaphore.
`trtWait`	Wait on a semaphore.
`trtSignal`	Signal a semaphore.

Table 2 The API of the real-time kernel.

3.4 API and Real-Time Primitives

The API of the real-time kernel is shown in Table 2. In addition to the initialization function and the function used to create tasks, the kernel supports a number of real-time primitives that may be called from the application programs. These include functions to retrieve the current global time, set and get the release and absolute deadline of a task, put a task to sleep until a certain time, and to terminate a task.

The `trtSleepUntil` call involves both setting the new release time and the new absolute deadline the task will have when it is awakened. This needs to be done in a single function, since these calls would otherwise individually change the state of the task and possibly cause context switches.

Counting semaphores has also been implemented in order to support task synchronization and communicating under mutual exclusion. A semaphore is represented by an 8-bit unsigned integer (see Listing 2), and the signal and wait operations basically correspond to incrementing and decrementing this counter. If a task does a wait on semaphore i with the counter being zero, the task is suspended and its state is set to $i + 1$, as described in Section 3.1. When a task does a signal on a semaphore, the task vector of the kernel is scanned for tasks waiting for this semaphore. Of these tasks, if any, the one with the shortest time to its deadline is made ready.

For the complete description and implementation of the various real-time primitives and the kernel initialization and task creation functions, see Appendix B.

4. An Application Example

The real-time kernel was used to implement concurrent control of two ball and beam laboratory processes. Three tasks were used to control each process, for a total of seven tasks (including the idle task). The controller was implemented using a cascaded structure with one task for the inner and one task for the outer loop of the cascade. Since the AVR only supported $+10$ or -10 analog output voltage, pulse width modulation was necessary to generate the desired control signal. This was implemented in software as two separate tasks.

Figure 2 The ball and beam laboratory process.

4.1 The Process

The ball and beam laboratory process is shown in Figure 2. The horizontal beam is controlled by a motor, and the objective is to balance the ball along the beam. The measurement signals from the system are the beam angle, denoted by ϕ, and the ball position on the beam, denoted by x. A linearized model of the system is given by

$$G(s) = G_\phi(s)G_x(s) \tag{1}$$

where

$$G_\phi(s) = \frac{k_\phi}{s} \tag{2}$$

is the transfer function between the motor input and the beam angle, and

$$G_x(s) = -\frac{k_x}{s^2} \tag{3}$$

is the transfer function between the beam angle and the ball position. The gains of the systems are given by $k_\phi \approx 4.4$ and $k_x \approx 9$.

4.2 The Controller

The cascaded controller is shown in Figure 3. The outer controller is a PID-controller and the inner controller is a simple P-controller. The outer controller was implemented according to the equations

$$\begin{aligned} D(k) &= a_d \cdot D(k-1) - b_d \cdot (y(k) - y(k-1)) \\ u(k) &= K \cdot (y_r - y(k)) + I(k) + D(k) \\ I(k+1) &= I(k) + a_i \cdot (y_r - y(k)) \end{aligned} \tag{4}$$

with the input signal, y, being the measured ball position and the output, u, being the reference angle for the inner P-controller. The a_d, b_d, and a_i parameters are given by $a_d = \frac{T_d}{T_d+Nh}$, $b_d = \frac{KT_DN}{T_d+Nh}$, and $a_i = \frac{Kh}{T_i}$.

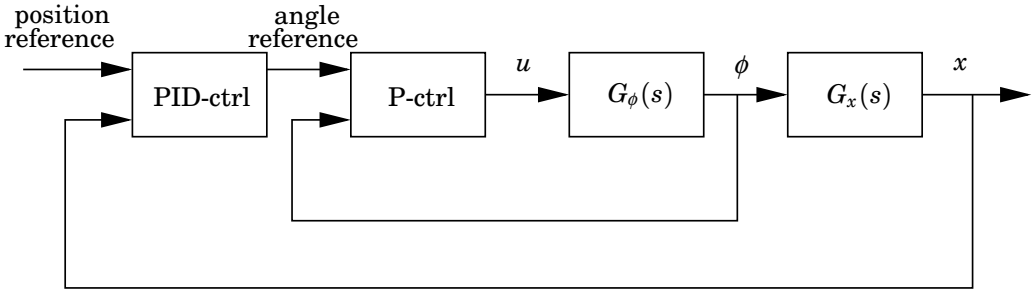

Figure 3 The cascaded controller structure for the ball and beam process.

The controller was implemented as a multi-rate controller, where one task was used for the inner loop and another task for the outer loop. The inner controller was running with a 20 ms sampling interval, whereas the outer controller used a 40 ms sampling interval. The controller parameters were chosen to $K_{inner} = 2$, $K_{outer} = -0.25$, $T_i = 10$, $T_d = 0.9$, and $N = 10$.

The controllers were implemented using fixed-point arithmetics with a representation using 5 integer bits and 11 fractional bits. The code executed by the two controller tasks are given by Listings 4 and 5 in Appendix A.

Since the angle reference is communicated between the outer and inner controller tasks, and the control signal is communicated between the inner controller and the PWM task, semaphores were used to guarantee mutual exclusion when accessing these variables.

4.3 The PWM Output Task

As seen in Listing 5, the control signal generated by the inner P-controller is an integer number in the interval $[-512, 511]$. This signal needs to be converted to an output in the interval $[-10, 10]$. However, the analog output channels can only generate $+10$ or -10 volt, depending on the value of the corresponding bit in the $PORTB$ register. Therefore, a pulse width modulation was implemented as given by Listing 6 in Appendix A.

The PWM task runs with a 128 tick cycle time (corresponding to 22 ms with the pre-scaler set to 256), outputting $+10$ volt in x ticks and -10 volt in $(128 - x)$ ticks. The x is determined from the desired control signal. E.g., to output 0 volt, x is chosen to 64.

4.4 Experiments

In the experiments six tasks were used to control two ball and beam processes. The main program for the experiment, were the tasks are created, is given by Listing 7 in Appendix A. Results of the experiments are shown in Figures 4–6 for one of the processes.

Two things can be noted from the plots. First, the integral action is quite slow, which is mainly due to the quantization in the control signal relative the increments of the I-part. Because of the large relative round-off error in the a_i-parameter of Equation 4, it was not possible to increase the integral further without jeopardizing the stability of the system during the transients.

Second, it can also be seen that the control signal is quite noisy. This is due to our implementation of the PWM, which is switching between $+10$ and -10 volts

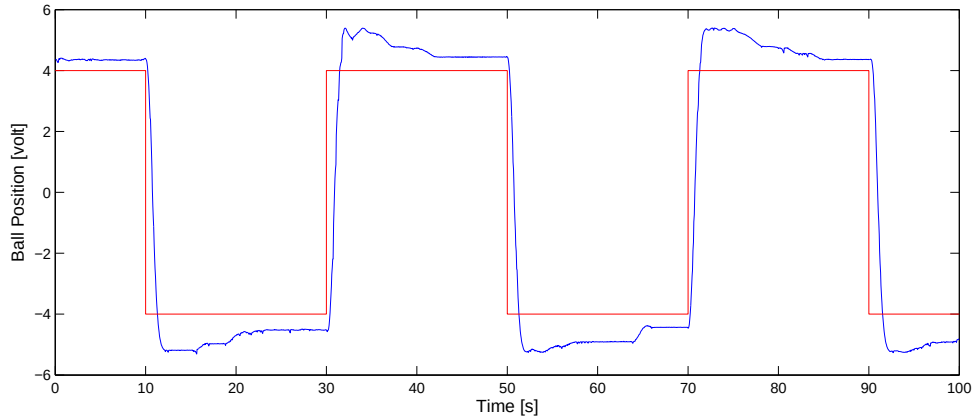

Figure 4 Ball position and reference during the experiment.

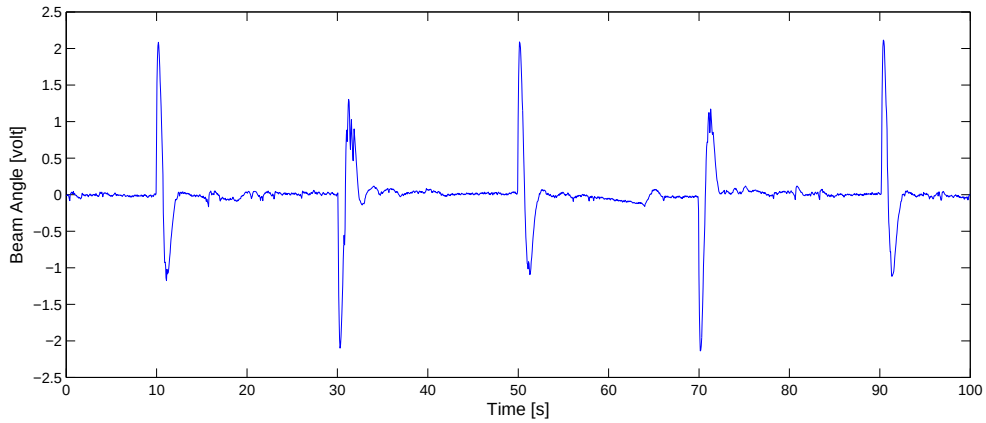

Figure 5 Beam angle during the experiment. Controlled by the inner P-controller.

with a quite slow frequency. The software implementation of the PWM output was done only to include more tasks, and to test the real-time kernel. Otherwise, a far superior option would have been to use the PWM functionality of the AVR hardware.

The serial communication device was used to monitor the kernel to find out how loaded the system is. The currently running task was written at a 115.2k Baud rate to sample the execution trace. The result is shown in Figure 7, and it can be seen that the load of the system is quite low. The approximate utilization was calculated to 10 per cent.

5. Conclusions

This report has described the design and implementation of an event-based real-time kernel for the Atmel ATmega8L AVR. The kernel uses earliest-deadline-first scheduling and supports ordinary counting semaphores for task synchronization.

The focus of the report has been on the memory management, timing, and internal workings of the kernel. An application example was also described, where

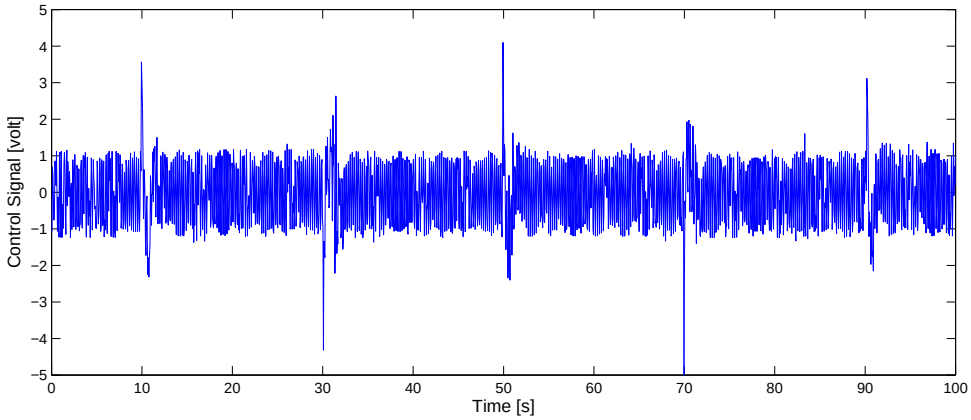

Figure 6 Control signal during the experiment. The high noise level is due to our slow realization of the PWM output. The desired output is generated by switching between −10 and +10 volt at a 1 kHz frequency.

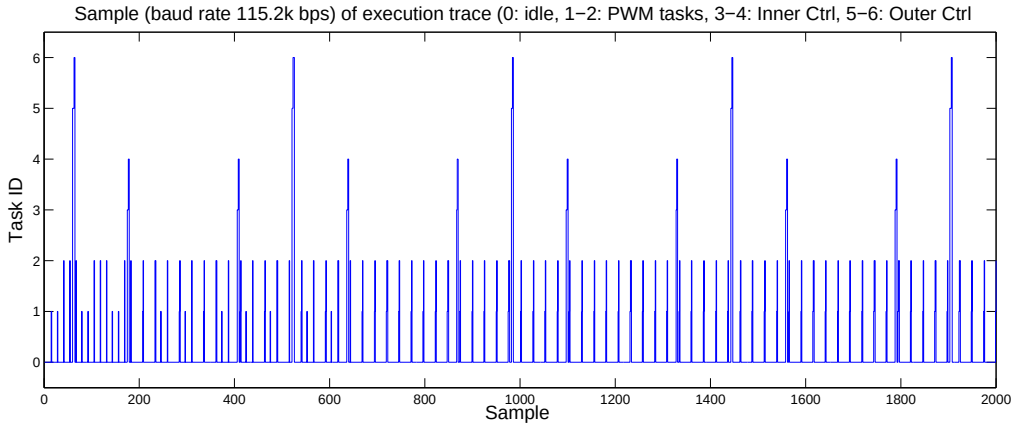

Figure 7 Sample of the execution trace during the experiment. The system utilization was calculated to around 10 per cent.

the real-time kernel was used to implement concurrent control of two ball and beam laboratory processes using six application tasks.

5.1 Lessons Learned

It turned out to be possible to implement an event-based EDF kernel with high timing resolution on a small embedded system such as the Atmel AVR. The kernel also has a relatively small memory footprint, occupying approximately 1200 bytes of flash memory. The RAM memory requirement depends on the number of tasks and semaphores in the system, and became around 100 bytes for the ball and beam application.

However, the kernel has its limitations, which was most clearly visible in the implementation of the PWM output in the ball and beam experiment. It turned out that the PWM operation was extremely jitter sensitive and could not be performed satisfactory with a high time resolution. This gave an indication of timing problems introduced by the kernel when running with high resolution. The kernel also does not implement cyclic timing and, thus, has a finite life time.

6. References

Atmel (2003): "Atmel AVR 8-bit RISC." Home page, http://www.atmel.com/products/AVR/.

Carlini, A. and G. Buttazzo (2003): "An efficient time representation for real-time embedded systems." In *Proceedings of the 2003 ACM Symposium on Applied Computing*, pp. 705–712. Melbourne, Florida.

A. Code Listings

Listing 3 The clock interrupt handler of the kernel.

```
SIGNAL(SIG_OUTPUT_COMPARE1A) {
  // Store r0-r31,SREG on the stack, done by the compiler

  if (TIFR & 0x04) { ++kernel.cycles; TIFR |= 0x04; }
  now = (kernel.cycles << 16) + TCNT1;

  // Release tasks from TimeQ and determine new running task
  for (i=1; i <= kernel.nbrOfTasks; i++) {
    t = &kernel.tasks[i];
    if (t->state == TIMEQ) {
      if (t->release <= now)
        t->state = READYQ;
      else if (t->release < nextHit)
        nextHit = t->release;
    }
    if (t->state == READYQ)
      if (t->deadline < kernel.tasks[running].deadline)
        running = i;
  }

  if (running != oldrunning) {
    // store old context
    t = &kernel.tasks[oldrunning];
    t->sp = SP;
    // load new context
    t = &kernel.tasks[running];
    SP = t->sp;
    kernel.running = running;
  }

  now = (kernel.cycles << 16) + TCNT1;
  timeleft = (int32_t)nextHit - (int32_t)now;
  if (timeleft < 4) {
    timeleft = 4;
  }

  if ((unsigned long)TCNT1 + timeleft < 65536) {
    OCR1A = TCNT1 + timeleft;
  } else if (TCNT1 < 65536 - 4) {
    OCR1A = 0x0000;
  } else {
    OCR1A = 4;
  }

  // Restore r0-r31,SREG from the stack;
}
```

Listing 4 Generic code executed by the two tasks implementing the outer PID-controllers. `args->angleRefPtr` is a pointer to a global variable for the angle reference, that is communicated between the outer and inner controller tasks. This global variable is protected by a semaphore.

```
void OuterCtrl(void *args) {

  /* Variable declarations omitted */

  args->t = trtGetRelease();
  args->d = trtGetDeadline();
  while(1) {

    // Set AnalogIn channel for position measurement
    ADMUX = args->pos_channel;

    // Read position sensor
    ADCSRA |= 0x40;         // start conversion
    while (ADCSRA & 0x40);  // wait for conversion to finish

    args->value = ADC;      // read value from A/D converter
    args->pos = (args->value - 512) << 5;

    // Calculate Output
    args->D = ((long)args->ad*(long)args->D - ...
               ... -  (long)args->bd*(long)(args->pos - args->yold)) >> 11;

    ref = *(args->ref); // pointer to global variable for ball reference
    args->P = ((long)args->K*(long)(ref - args->pos)) >> 11;

    trtWait(args->mutex); // Semaphore protecting global variable
    *(args->angleRefPtr) = args->P + args->I + args->D;
    trtSignal(args->mutex);

    // Update State
    args->I = args->I + (int)(((long)args->ai*(long)(ref - args->pos)) >> 11);
    args->yold = args->pos;

    // Sleep, 40 ms sampling interval
    args->t += SECONDS2TICKS(0.04);
    args->d += SECONDS2TICKS(0.04);
    trtSleepUntil(args->t, args->d);
  }
}
```

Listing 5 Generic code executed by the two tasks implementing the inner P-controllers. `args->angleRefPtr` and `args->uPtr` are pointers to global variables for the angle reference and the control signal, that are communicated between the outer and inner controller and the inner controller and the PWM task, respectively. These global variables are protected by semaphores.

```
void InnerCtrl(void *args) {

  /* Variable declarations omitted */

  args->t = trtGetRelease();
  args->d = trtGetDeadline();
  while(1) {

    // Set AnalogIn channel for angle measurement
    ADMUX = args->ang_channel;

    // Read angle sensor
    ADCSRA |= 0x40;         // start conversion
    while (ADCSRA & 0x40);  // wait for conversion to finish

    args->value = ADC;      // read value from A/D converter
    args->angle = (args->value - 512) << 5;

    trtWait(args->mutex1);    // Semaphore protecting global variable
    angleRef = *(args->angleRefPtr);
    trtSignal(args->mutex1);

    args->u = (-args->K*(angleRef - args->angle)) >> 5;

    trtWait(args->mutex2);    // Semaphore protecting global variable
    *(args->uPtr) = args->u;
    // limit control
    if (args->u > 511) {
      *(args->uPtr) = 511;
    } else if (args->u < -512) {
      *(args->uPtr) = -512;
    }
    trtSignal(args->mutex2);

    // Sleep, 20 ms sampling interval
    args->t += SECONDS2TICKS(0.02);
    args->d += SECONDS2TICKS(0.02);
    trtSleepUntil(args->t, args->d);
  }
}
```

Listing 6 Generic PWM code used to generate the desired output signals for the two ball and beam processes. `args->uPtr` is a pointer to a global variable for the control signal, that is communicated between the inner controller task and the PWM task. This global variable is protected by a semaphore.

```
void PWMtask(void* args) {

  /* Variable declarations omitted */

  args->t = trtGetRelease();
  while (1) {
    PORTB |= args->channel; // Output +10 volt

    trtWait(args->mutex);     // Semaphore protecting global variable
    // Sleep for x ticks
    args->t += ((uint16_t) (*(args->uPtr) + 512)) >> 3;
    trtSignal(args->mutex);

    trtSleepUntil(args->t, 0);

    PORTB &= ~args->channel; // Output -10 volt

    trtWait(args->mutex);     // Semaphore protecting global variable
    // Sleep for (128-x) ticks
    args->t += 128 - (((uint16_t) (*(args->uPtr) + 512)) >> 3);
    trtSignal(args->mutex);

    trtSleepUntil(args->t, 0);
  }
}
```

Listing 7 The main program for the ball and beam application.

```
int main(void) {
  uint8_t i;

  ADCSRA = 0xc7;  // ADC enable

  // Initialize kernel
  trtInitKernel(80);


  // 4 semaphores to protect common variables
  for (i=1; i <= 4; i++) {
    trtCreateSemaphore(i, 1); // Semnbr i, Initval 1
  }


  // BaB System 1, inner and outer ctrl released with offset of 10 ms

  trtCreateTask(PWMtask, 100, SECONDS2TICKS(0.0), SECONDS2TICKS(0));
  trtCreateTask(InnerCtrl, 100, SECONDS2TICKS(0.01), SECONDS2TICKS(0.02));
  trtCreateTask(OuterCtrl, 150, SECONDS2TICKS(0.0),  SECONDS2TICKS(0.04));


  // BaB System 2, released with an offset of 5 ms to controller 1

  trtCreateTask(PWMtask, 100, SECONDS2TICKS(0.0), SECONDS2TICKS(0));
  trtCreateTask(InnerCtrl, 100, SECONDS2TICKS(0.015), SECONDS2TICKS(0.02));
  trtCreateTask(OuterCtrl, 150, SECONDS2TICKS(0.005), SECONDS2TICKS(0.04));


  // Idle task
  while (1);

}
```

B. Command Reference

trtInitKernel

Purpose

Initialize the real-time kernel.

Syntax

```
void trtInitKernel(uint16_t idletask_stack)
```

Arguments

`idletask_stack` Stack size to be reserved for the idle task.

Description

This function performs necessary initialization steps of the kernel data structure, and must, therefore, be called first of all in the main program.

Implementation

```
void trtInitKernel(uint16_t idletask_stack) {

  /* Set up timer 1 */
  TCNT1 = 0x0000;        /* reset counter 1 */
  TCCR1A = 0x00;         /* normal operation */
  TCCR1B = 0x04;         /* prescaler = 256 */
  TIMSK = BV(OCIE1A);    /* Enable compare match */

  kernel.memptr = (void*)(RAMEND - idletask_stack);
  kernel.nbrOfTasks = 0;
  kernel.running = 0;

  kernel.cycles = 0x0000;
  kernel.nextHit = 0x7FFFFFFF;

  // Initialize idle task (task 0)
  kernel.tasks[0].deadline = 0x7FFFFFFF;
  kernel.tasks[0].release = 0x00000000;

  sei(); /* set enabled interrupts */
}
```

trtCreateTask

Purpose

Create a task.

Syntax

```
void trtCreateTask(void (*fun)(void*), uint16_t stacksize,
                   uint32_t release, uint32_t deadline, void *args)
```

Arguments

`fun`	The address (function pointer) of the code function specifying the execution of the task.
`stacksize`	Stack size to be reserved for the task.
`release`	The release offset of the task specified in timer ticks.
`deadline`	The relative deadline of the task, specified in timer ticks.
`args`	An arbitrary data structure for task specific arguments. Given as input argument to the code function.

Description

This function is used to create a task to run in the real-time kernel. This function may be called either at start-up from the main program or during run-time from the application to create new tasks dynamically. It is up to the application programmer to specify sufficient stack memory and to make sure that the SRAM memory is not used up. Only a limited number of tasks may be created, as specified by the user in the pre-defined variable `MAXNBRTASKS`. The code function, `fun`, is typically implemented as an infinite loop, and has a task-specific input argument `args` that facilitates using the same code function for several tasks (see, e.g., the generic code functions in Listings 4–6 in Appendix A).

Implementation

```
void trtCreateTask(void (*fun)(void), uint16_t stacksize,
                   uint32_t release, uint32_t deadline, void *args) {

  uint8_t *sp;
  struct task *t;
  uint8_t i;

  cli(); // disable interrupts

  ++kernel.nbrOfTasks;

  sp = kernel.memptr;
  kernel.memptr -= stacksize;  // decrease free mem ptr

  // initialize stack
  *sp-- = lo8(fun);        // store PC(lo)
  *sp-- = hi8(fun);        // store PC(hi)
  for (i=0; i<24; i++)
```

```
    *sp-- = 0x00;          // used to store SREG, r0-r23

  // Save args in r24-25 (input arguments stored in these registers)
  *sp-- = lo8(args);
  *sp-- = hi8(args);

  for (i=0; i<6; i++)
    *sp-- = 0x00;         // store r26-r31

  // initialize task attributes
  t = &kernel.tasks[kernel.nbrOfTasks];

  t->release = release;
  t->deadline = deadline;
  t->state = TIMEQ;    // put task in time queue

  t->sp = sp;          // store stack pointer

  SIG_OUTPUT_COMPARE1A(); // Call interrupt handler to schedule task
                          // This is the clock interrupt handler that
                          // executes the kernel during run-time

}
```

trtTerminate

Purpose

Terminate the execution of the current task.

Syntax

```
void trtTerminate(void)
```

Description

This function terminates the execution of the currently running task, by setting its state to `TERMINATED`.

Implementation

```
void trtTerminate(void) {

  cli(); // disable interrupts

  kernel.tasks[kernel.running].state = TERMINATED;

  SIG_OUTPUT_COMPARE1A(); // call interrupt handler to schedule
}
```

trtCurrentTime

Purpose

Get the current system time.

Syntax

```
uint32_t trtCurrentTime(void)
```

Description

This function returns the current time since system start. The return value is in timer ticks, where the time of each tick depends on the pre-scaler factor, N, according to

$$TICKTIME = N \cdot 14.7456 \cdot 10^{-6} \text{ seconds}$$

Implementation

```
uint32_t trtCurrentTime() {

  return (((uint32_t)kernel.cycles << 16) + (uint32_t)TCNT1);
}
```

trtSleepUntil

Purpose

Put a task to sleep until a certain time.

Syntax

```
void trtSleepUntil(uint32_t release, uint32_t deadline)
```

Arguments

`release`	The time when the task should wake up.
`deadline`	The new absolute deadline of the task.

Description

This function is used to make a task sleep until a specified point in time. The absolute deadline of the task is also updated using this function.

Implementation

```
void trtSleepUntil(uint32_t release, uint32_t deadline) {

  struct task *t;

  t = &kernel.tasks[kernel.running];

  cli(); // turn off interrupts

  t->state = TIMEQ;
  t->release = release;
  t->deadline = deadline;

  SIG_OUTPUT_COMPARE1A(); // call interrupt handler to schedule
}
```

trtGetRelease

Purpose

Retrieve the latest release time of the calling task.

Syntax

```
uint32_t trtGetRelease()
```

Description

This primitive is used to retrieve the latest release time of the calling task. The return value is given in timer ticks.

Implementation

```
uint32_t trtGetRelease() {

  return kernel.tasks[kernel.running].release;
}
```

trtGetDeadline

Purpose

Retrieve the absolute deadline of the calling task.

Syntax

```
uint32_t trtGetDeadline()
```

Description

This primitive is used to retrieve the absolute deadline of the calling task. The return value is given in timer ticks.

Implementation

```
uint32_t trtGetDeadline() {

  return kernel.tasks[kernel.running].deadline;
}
```

trtCreateSemaphore

Purpose

Initialize a semaphore.

Syntax

```
void trtCreateSemaphore(uint8_t semnbr, uint8_t initval)
```

Arguments

`semnbr`	The number used to identify the semaphore. Should be between 1 and `MAXNBRSEMAPHORES`.
`initval`	The initial value of the semaphore counter.

Description

This primitive is used to initialize a semaphore, and must be called before any corresponding `trtWait` or `trtSignal` calls.

Implementation

```
void trtCreateSemaphore(uint8_t semnbr, uint8_t initVal) {

  cli(); // turn off interrupts

  kernel.semaphores[semnbr-1] = initVal;

  sei(); // set enabled interrupts;
}
```

trtWait

Purpose

Wait for a semaphore.

Syntax

```
void trtWait(uint8_t semnbr)
```

Arguments

`semnbr`	The number used to identify the semaphore. Should be between 1 and `MAXNBRSEMAPHORES`.

Description

This primitive is used to attempt to take a semaphore. If no other task is holding the semaphore, execution of the calling task continues and the semaphore counter is decremented. Otherwise, the state of the calling task is set to `WAIT_OFFSET + semnbr`, and a context switch will take place.

Implementation

```
void trtWait(uint8_t semnbr) {

  struct task *t;
  unsigned char *s;

  t = &kernel.tasks[kernel.running];

  cli(); // disable interrupts

  s = &kernel.semaphores[semnbr-1];
  if ((*s) > 0) {
    (*s)--;
  } else {

    t->state = semnbr + WAIT_OFFSET; // waiting for Sem #semnbr
    SIG_OUTPUT_COMPARE1A(); // call interrupt handler to schedule
  }

  sei(); // reenable interrupts
}
```

trtSignal

Purpose

Signal a semaphore.

Syntax

```
void trtSignal(uint8_t semnbr)
```

Arguments

`semnbr`	The number used to identify the semaphore. Should be between 1 and `MAXNBRSEMAPHORES`.

Description

This primitive is used to signal that a semaphore is released. The highest-priority task with state equal to `WAIT_OFFSET + semnbr` will then be made ready and a context switch may occur.

Implementation

```
void trtSignal(uint8_t semnbr) {

  uint8_t i;
  struct task *t;
  uint32_t minDeadline = 0xFFFFFFFF;
  uint8_t taskToReadyQ = 0;

  cli(); // disable interrupts

  for (i=1; i <= kernel.nbrOfTasks; i++) {
    t = &kernel.tasks[i];
    if (t->state == (semnbr + WAIT_OFFSET)) {
      if (t->deadline <= minDeadline) {
        taskToReadyQ = i;
        minDeadline = t->deadline;
      }
    }
  }

  if (taskToReadyQ == 0) {
    kernel.semaphores[semnbr-1]++;
  } else {
    kernel.tasks[taskToReadyQ].state = READYQ; // make task ready
    SIG_OUTPUT_COMPARE1A(); // call interrupt handler to schedule
  }

  sei(); // reenable interrupts
}
```