Multirate Feedback Control Using the TinyRealTime Kernel

Henriksson, Dan; Cervin, Anton

Published in:
Computer and Information Sciences - ISCIS 2004 19th International Symposium, Proceedings (Lecture Notes in Computer Science)

DOI:
10.1007/b101749

2004

Citation for the published paper:
Henriksson, Dan and Cervin, Anton, 2004,
"Multirate Feedback Control Using the TinyRealTime Kernel",
Computer and Information Sciences - ISCIS 2004 19th International Symposium, Proceedings (Lecture Notes in Computer Science)
ISBN: 978-3-540-23526-2. Publisher: Springer

19th International Symposium on Computer and Information Sciences, Antalya, Turkey, October 2004

Multirate Feedback Control Using the TINYREALTIME Kernel

Dan Henriksson, Anton Cervin

Department of Automatic Control
Lund Institute of Technology
Box 118, SE-221 00 Lund, Sweden
{dan,anton}@control.lth.se

Abstract

Embedded microcontrollers are often programmed in plain C and lack support for multithreading and real-time scheduling. This can make it very cumbersome to implement multirate feedback control applications. We have developed the TINYREALTIME kernel for the Atmel ATmega8L AVR to show that it is feasible to use high-precision, deadline-based scheduling even in a tiny 8-bit processor with 1 KB of RAM. The kernel is demonstrated in a multirate control application, where six periodic real-time tasks (four control tasks and two pulse width modulation tasks) are used to control two ball-and-beam processes.

1. Introduction

The growing complexity of embedded real-time control applications has increased the need for kernel support for multiprogramming and dynamic real-time scheduling even in tiny embedded systems. Traditionally, embedded systems have been programmed in plain C, using interrupt routines to handle the time-critical operations. This approach works fine as long as there is only one activity with strict timing requirements in the system. For multirate control applications, however, such an approach can quickly become very cumbersome. We argue that the convenient techniques of concurrent programming and real-time scheduling are feasible even for tiny embedded systems.

In this paper, we describe the architecture and application of TINYREALTIME [8], a tiny real-time kernel for the Atmel ATmega8L 8-bit AVR microcontroller. The ATmega8L features a RISC processor with up to 16 MIPS throughput at 16 MHz, 8 KB flash program memory, and 1 KB SRAM. For timing, the microcontroller features one 16-bit timer/counter and two 8-bit timers/counters with separate prescalers. The ATmega8L has a rich I/O interface, including a 6-channel 10-bit A/D converter, 23 programmable digital I/O ports, three PWM channels, and various serial communication interfaces. For a more detailed description of the ATmega8L AVR, see [2].

The TINYREALTIME kernel is event-triggered and implements fully preemptive earliest-deadline-first (EDF) task scheduling. Counting semaphores are provided to support task synchronization. The Stack Resource Protocol (SRP) [3] will be implemented as resource access policy.

The memory requirement of the kernel is small, with a flash memory footprint of approximately 1200 bytes. It further occupies 11 bytes of RAM for the kernel data structure plus an additional 11 bytes for each task and one byte for each semaphore.

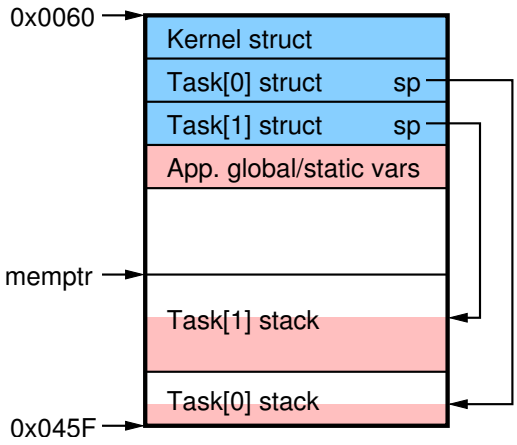

Figure 1 Memory layout of the real-time kernel.

A multirate real-time control application has been developed to demonstrate the feasibility of the real-time kernel. The application involves two ball-and-beam laboratory processes which are concurrently controlled using six application tasks. Each controller is implemented using a cascaded structure with two tasks running at different rates. Two additional tasks are used to implement a simple pulse width modulation of the output signal. The experiments display satisfactory control performance, good timing behavior, and low CPU utilization of the target system.

Related Work

There exist dozens of commercial and non-commercial real-time kernels for small embedded systems that support either cooperative, round-robin, or priority-preemptive scheduling. These kernels do not, however, support explicit timing constraints in the form of deadlines and dynamic-priority scheduling. One noticeable exception is ERIKA Enterprise [6], a commercial variant of the open-source kernel ERIKA Educational kernel [7]. ERIKA Enterprise supports several scheduling algorithms and resource protocols, including fully preemptive EDF with the stack resource protocol. It has a small ROM footprint (about 1600 bytes), making it suitable for tiny embedded systems. Supported targets include the Hitachi H8 and the ARM 7.

2. Kernel Overview

Memory Layout and Data Structures

The ATmega8L AVR has 1120 memory locations, of which the first 96 bytes are used for the register file and the I/O memory, and the following 1024 bytes represent the internal RAM. In the kernel implementation, the 1024 bytes of RAM are utilized according to the memory layout in Figure 1, which shows the location of the kernel and task data structures and the individual stack memories. The kernel data structure and the task data structure are given by Listings 1 and 2.

As seen in Figure 1, the kernel and task data structures are allocated statically 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.

Listing 1 The kernel data structure.

```
#define MAXNBRTASKS ...
#define MAXNBRSEMAPHORES ...

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

Listing 2 The task data structure.

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

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 2), and the next wake-up time of the kernel.

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, n_1, and the maximum number of semaphores, n_2, the total RAM memory requirement of the kernel is $11 + 11n_1 + n_2$.

Timing

The output compare match mode of the 16-bit timer 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.

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.

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

Prescaler	*Clock resolution*	*System 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

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

The problem with limited life time versus high timing resolution can be avoided by using a circular clock implementation [4]. 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).

Kernel Internal Workings

The kernel is implemented in the interrupt handler associated with the output compare match interrupt of the timer. When the interrupt handler is entered, the status register and the 32 working registers are stored on the stack of the currently running task.

Thereafter, the task vector is traversed in order to determine if any tasks should be released at the current time. In accordance with EDF, the ready task with the closest absolute deadline is then made the running task, which may trigger a context switch. In that case, the current address of the stack pointer is stored in the task struct associated with the preempted task, and the stack pointer is updated with the corresponding value of the new running task.

Finally, a new clock interrupt is set up, by updating the output compare match register. The clock interrupt is set to the closest release time among the sleeping tasks.

API and Real-Time Primitives

The API of the TINYREALTIME kernel is shown in Table 2. `trtInitKernel` is used to initialize the kernel and must be called first of all from the main program. Tasks are then created by the function `trtCreateTask`. Here the user specifies the code function to be executed, stack size, release offset, deadline, and an arbitrary data structure for the task. Tasks may be terminated using the function `trtTerminate`.

The implementation also provides 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 wait for and signal semaphores.

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.

Table 2 The API of the TINYREALTIME kernel.

Command	*Description*
`trtInitKernel`	Initialize the kernel.
`trtCreateTask`	Create a task, specifying its release time and absolute deadline.
`trtTerminate`	Terminate the execution of the current task.
`trtCurrentTime`	Get the current system time.
`trtSleepUntil`	Put the running task to sleep, specifying new release and deadline.
`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.

Counting semaphores has also been implemented in order to support task synchronization and communication under mutual exclusion. A semaphore is represented by an 8-bit unsigned integer (see Listing 1), 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 2. 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 a complete description and implementation details of the various real-time primitives and the kernel initialization and task creation functions, see [8].

3. Multirate Feedback Control

The TINYREALTIME 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 supports digital output, pulse width modulation (PWM) was necessary to generate the desired control signal. This was implemented in software as two separate tasks, one for each control loop.

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

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

Figure 2 The ball-and-beam laboratory process.

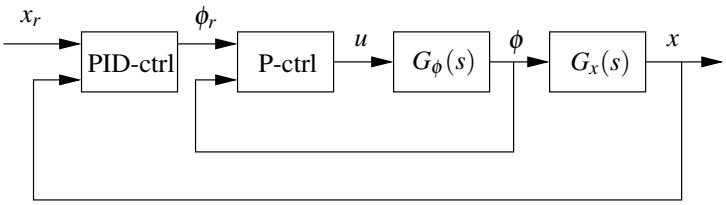

Figure 3 The cascaded controller structure for the ball-and-beam process.

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 $k_x \approx 9$.

The Controller

The structure of 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

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

with the input signal, y, being the measured ball position and the output, u, being the reference angle for the inner P-controller. The parameters a_d, b_d, and a_i are precomputed and are given by

$$a_d = \frac{T_d}{T_d + Nh} \quad b_d = \frac{KT_D N}{T_d + Nh} \quad a_i = \frac{Kh}{T_i}$$

The controller was implemented as a multirate 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, see Figure 4. The controller parameters were chosen to $K_{inner} = 2$, $K_{outer} = -0.25$, $T_i = 10$, $T_d = 0.9$, and $N = 10$, giving the lumped controller parameters $a_d = 0.692$, $b_d = 1.731$, $a_i = 0.004$. The relatively large difference in magnitude of the parameters affected the implementation, which was performed by fixed-point arithmetics with a representation using 5 integer bits and 11 fractional bits.

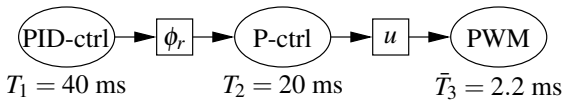

Figure 4 Communicating tasks and shared variables in the multirate structure.

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, as shown in Figure 4. Global variables were used for the communication and two semaphores were created for mutual exclusion when accessing the common variables.

Pulse Width Modulation

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 output channels can only generate $+10$ or -10 volt, depending on the value of the corresponding bit in the *PORTB* register of the AVR.

To solve this problem, a pulse width modulation task was implemented for each control loop. The PWM task runs with a 128 tick cycle time (corresponding to 2.2 ms with the prescaler 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.

Experiments

In the experiments six tasks were used to control two ball-and-beam processes concurrently. Results of the experiments are shown in Figures 5–7 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 with a quite slow frequency. The software implementation of the PWM output was done only to include more tasks, for the purpose of stressing the real-time kernel. Otherwise, a far superior option would have been to use the PWM functionality of the AVR hardware.

The kernel was monitored using the serial communication to estimate the system load. The currently running task was written at a 115.2k Baud rate to sample the execution trace. The result is shown in Figure 8, and it can be seen that the load of the system is quite low. The approximate utilization was calculated to 10 percent.

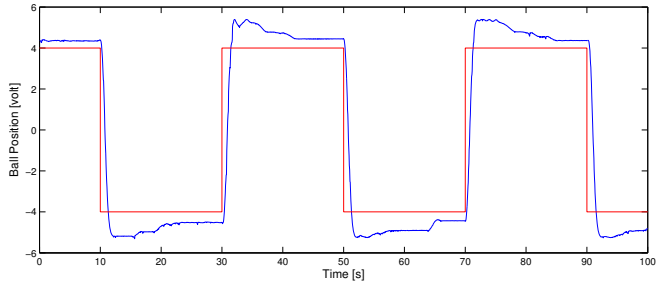

Figure 5 Ball position and reference during the experiment.

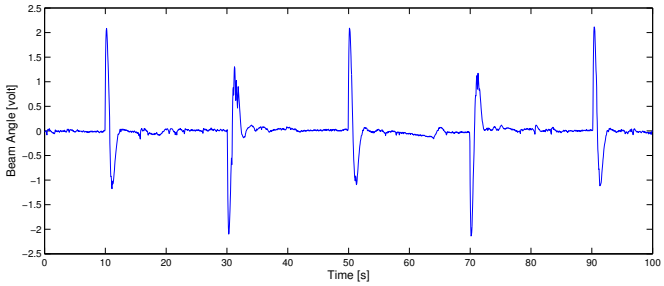

Figure 6 Beam angle during the experiment.

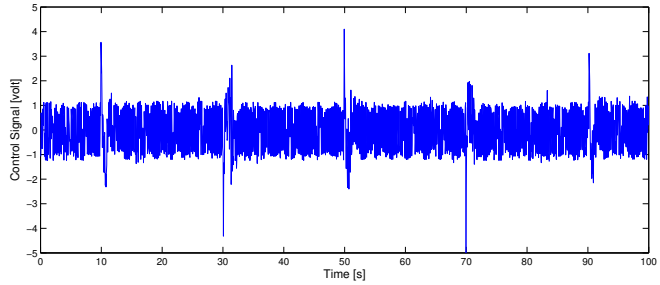

Figure 7 Control signal during the experiment.

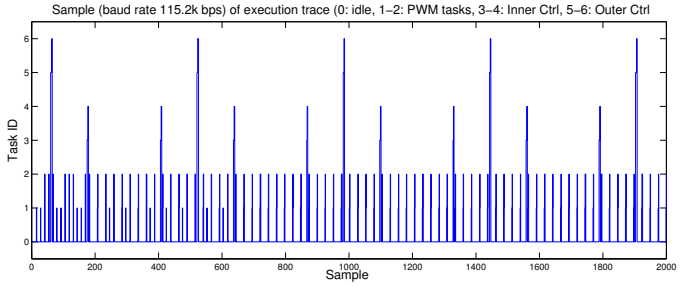

Figure 8 Sample of the execution trace during the experiment.

4. Conclusions

The paper has described the design and application of TINYREALTIME, an event-triggered real-time kernel for an Atmel AVR 8-bit micro-controller. The kernel supports multiprogramming by means of dynamic deadline-based scheduling of tasks and counting semaphores for synchronization. The kernel footprint is small ($\approx$ 1200 bytes).

The kernel was evaluated in a real-time control application involving six user tasks. The application demonstrated the feasibility of implementing event-based deadline-driven scheduling with high timing resolution on a tiny embedded system such as the Atmel AVR. The RAM memory requirement for the kernel and task data structures in the application was about 100 bytes.

Future Work

The TINYREALTIME kernel may be extended in many ways. The current implementation can be made more efficient, and because of the relatively low memory requirement of the current implementation a lot of additional functionality may be implemented. Some of the things that will be implemented in the future are summarized below.

The current version of the kernel was written specifically for the AVR 8-bit RISC platform. However, it is desirable to have a kernel architecture that supports other platforms as well. This would require the definition of a hardware abstraction layer for the hardware-dependent routines, such as interrupt and context switch handling.

The cyclic timer mentioned in Section 2 will be implemented. This will allow a high timing resolution and an infinite system life time. It will also increase the performance of the kernel by reducing the number of bits used in the timing representation.

Currently, no resource access protocol is implemented for the semaphores. To this end we intend to implement the Stack Resource Protocol (SRP) [3]. This is an extension to EDF-scheduled systems of the Priority Ceiling Protocol [9] that provides low worst-case blocking times. We will also implement higher-level synchronization mechanisms, such as monitors with condition variables.

An additional advantage of SRP is that it guarantees that once a task starts executing, it will not be blocked until completion. This means that a lower-priority task never will be executed before the task is completed. This allows for a more efficient implementation where all tasks in the system can share the same single stack.

Finally, we intend to implement kernel primitives to handle execution time budgets for tasks. This would facilitate EDF-based server scheduling, such as, e.g., the Constant Bandwidth Server [1], and the related Control Server mechanism [5] for real-time control system co-design.

5. References

[1] L. Abeni and G. Buttazzo. “Integrating multimedia applications in hard real-time systems.” In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, 1998.

[2] Atmel. “AVR 8-Bit RISC.” http://www.atmel.com/products/AVR, 2004.

[3] T. P. Baker. “Stack-based scheduling of real-time processes.” *Journal of Real-time Systems*, 1991.

[4] A. Carlini and G. Buttazzo. “An efficient time representation for real-time embedded systems.” In *Proc. ACM Symposium on Applied Computing*, 2003.

[5] A. Cervin and J. Eker. “Control-scheduling codesign of real-time systems: The control server approach.” *Journal of Embedded Computing*, 2004. To appear.

[6] Evidence. “ERIKA Enterprise.” http://www.evidence.eu.com/Erika.asp, 2004.

[7] P. Gai, G. Lipari, and M. Di Natale. “A flexible and configurable real-time kernel for time predictability and minimal RAM requirements.” Technical Report RETIS TR 2001-02, ReTiS Lab, Scuola Superiore S. Anna, Pisa, Italy, 2001.

[8] D. Henriksson and A. Cervin. “TinyRealTime—An EDF kernel for the Atmel ATmega8L AVR.” Technical Report ISRN LUTFD2/TFRT--7608--SE, Department of Automatic Control, Lund Institute of Technology, Sweden, February 2004.

[9] L. Sha, R. Rajkumar, and J. Lehoczy. “Priority inheritance protocols: An approach to real-time synchronization.” *IEEE Transactions on Computers*, 1990.