ARTK: a compact real-time kernel for Arduino

Paul H. Schimpf

Department of Computer Science,
Eastern Washington University,
Cheney, WA 99004, USA
E-mail: pschimpf@ewu.edu

Abstract: This article describes ARTK, a compact real-time kernel for the Arduino embedded systems development environment. It provides a priority-driven preemptive task scheduler, semaphores for task synchronisation, and serial channel output. It is compatible with Arduino boards containing more or less than 64 kbytes of memory. The memory footprint of ARTK is under 8 kbytes.

Keywords: embedded systems; real-time kernel; Arduino.

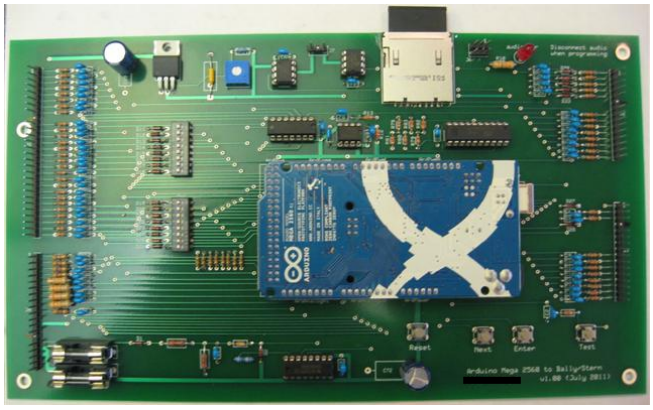
Reference to this paper should be made as follows: Schimpf, P.H. (2013) 'ARTK: a compact real-time kernel for Arduino', *Int. J. Embedded Systems*, Vol. 5, Nos. 1/2, pp.106–113.

Biographical notes: Paul H. Schimpf received his BSEE (Summa Cum Laude), MSEE, and PhD degrees from the University of Washington, Seattle, in 1982, 1987, and 1995, respectively. He began his academic career in 1998, and is currently the Chair of the Department of Computer Science at Eastern Washington University in Cheney, WA, USA. His research interests include numerical methods for forward and inverse solutions to partial differential equations, with biomedical applications. Prior to his academic career, he was employed as a Senior Principal Design Engineer in the electronics industry, where he enjoyed 15 years of experience developing parallel embedded signal and image processing systems.

1 Introduction

The Arduino (2012) provides a capable and extremely easy to use development environment for a wide variety of embedded systems projects. As an example, the author has developed an Arduino daughterboard (a *shield* in Arduino terminology) that can be substituted for the processor board in any of 85 different pinball machines manufactured by the Bally and/or Stern corporations (Figure 1). This shield provides processor interfaces to the switch matrix, lamps, score displays, and machine solenoids, and also adds the capability to play sound effect clips stored on an SD card through an on-board audio amplifier. The machines supported by this shield originally used a 6800 microprocessor, for which in-circuit emulators (ICE) are now nearly impossible to obtain. Without an ICE, code development for such machines requires extraction, burning, and insertion of an EPROM for each trial run, a process that is not conducive to debugging or suitable for an educational environment. Substitution of an Arduino processor has allowed the author to include a practical pinball machine programming project in an embedded systems course. This exercise integrates course topics such as interrupt service routines (ISRs), reading switch matrices, software switch de-bouncing, display multiplexing, and control of power devices (e.g., lamps and solenoids). The addition of a small and simple to use real-time operating system (RTOS) allows this exercise to also integrate course material on typical RTOS interfaces and real-time scheduling strategies such as rate-monotonic (Sha and Goodenough, 1990) and deadline monotonic scheduling (Tindell, 2000). At the time of this writing there is one other RTOS available for Arduino (Duinos, 2011).

Figure 1 Arduino mega to bally/stern pinball shield (see online version for colours)



ARTK was inspired by and has some structural similarities to Tempo, a kernel for small model 8086 programmes used in Buhr and Bailey (1999). ARTK is released under the GNU General Public License and may be obtained at http://penguin.ewu.edu/~pschimpf/pubfiles/ or by contacting the author. The author requests that the ARTK start-up message printing the author's name be left in any code modifications and that any publication mentioning a

system that makes use of ARTK, or any modification of ARTK, refer to this article.

2 ARTK software developer interface

ARTK uses a threading model to implement multi-tasking, and thus ARTK tasks have a unique stack and register space, but share global memory space with other tasks. The ARTK library is written in C++, but provides a C-like function interface to the user programme since Arduino programmes are typically written in a C style. ARTK provides the Arduino loop() and setup() functions, and asks the user to create a function called Setup() in lieu of the usual setup(). Tasks are typically created in Setup() although they may also be created by other tasks. Setup() is called prior to the start of multitasking, so any tasks created in Setup() do not gain an opportunity to run until after Setup() returns. When a task is created by another task it gains an opportunity to run when the creating task is preempted by a higher priority task, or yields to a task at the same or lower priority. A task may yield the processor in several ways: by exiting, by explicitly yielding, by sleeping, or by blocking on a semaphore.

Semaphores may also be created by a task or in the Setup() function. If an ISR that signals a semaphore is attached in Setup(), then that semaphore should also be created in Setup() in case the ISR fires right away. It is safe to signal a semaphore prior to the start of multitasking.

ARTK allows multiple tasks at the same priority level, but the processor is not time-sliced between them as in some RTOS's. When multiple tasks exist at the same priority level, they are rescheduled for execution relative to each other in a round-robin order anytime there is a context switch from one of those tasks. Thus a task moves to the end of the order for tasks at the same priority level when it is either preempted by a higher priority task or yields to a task at the same or lower priority by one of the mechanisms stated above. It should be noted that ARTK does not protect against priority inversions. If all tasks return from their root functions, the kernel exits with a message. The user can also shut down multitasking with a function call.

ARTK also provides a global mutual-exclusion semaphore (mutex) that can be acquired and released by surrounding a code block with the CS() macro. No block of code enclosed by CS() will be allowed to intermix execution with any other block of code enclosed by CS(). This can be used to enforce exclusive access to a shared resource, such as the serial output channel. The user can create additional groups of mutually exclusive execution using user-defined semaphores.

The ARTK developer interface functions are accessed by including the file ARTK.h in an Arduino programme (a *sketch* in Arduino terminology). A description of the ARTK function interface defined in that file is provided in Table 1.

Table 1 ARTK programmer interface

Function prototype	*Description*
`void Printf(char *fmt, ... )`	Provides C-language printf() functionality to the Arduino serial monitor interface.
`void ARTK_SetOptions( int large, int usec)`	Modify ARTK options. Any argument of -1 leaves that setting unchanged, and at the default if it has not been previously changed. This function should only be called from the Setup() function, which is called prior to ARTK startup of the task scheduler. The first argument, when non-zero, sets the context switcher for compatibility with Atmel memory spaces larger than 64kbytes (e.g., Arduino Mega processors). When 0 the context switcher is compatible with memory spaces of 64kbytes or less). The default is 0 (small memory model). Any change to the memory model MUST be made prior to the creation of any tasks. The second argument establishes the sleep and timed-wait timer interval. The default is 10 msec.
`CS(x)`	Critical Section. Code block x will not be executed until a semaphore shared by all such critical sections is obtained. Thus no section of code encapsulated in CS() will intermix execution with any other code section encapsulated by CS(). This may be useful, for example, in preventing the intermixing of multiple output streams.
`TASK ARTK_CreateTask( void (*root_fn)(), int priority = 1, unsigned int stacksize = 1024)`	Create a task with the passed root function, priority (defaults to 1), and stacksize (defaults to 256). These defaults can be changed by editing ARTK.h. Valid priority levels are 1 through 16. Values passed outside of that range are railed to the minimum or maximum. The minimum stack size is 256. Values smaller than that will be railed to 256.

Table 1 ARTK programmer interface (continued)

Function prototype	*Description*
`int ARTK_StackLeft()`	A task can call this function to determine how much stack space it has remaining. This function can be useful for debugging errant tasks. It should be noted that 33 bytes of stack will be consumed in order to save the processor state when the task is swapped out during a context switch.
`int ARTK_EstAvailRam()`	Estimate the amount of free RAM. This is a conservative estimate as it does not traverse the heap freelist. ARTK tasks consume 6–8 bytes of heap while sleeping, which is returned to the heap when they wake. If the current number of sleeping tasks is less than the maximum that have simultaneously slept, then there will be some amount free space on the heap that is not counted by this function. This function can be useful for debugging errant programs.
`TASK ARTK_MyId()`	Tasks that share a common root function can make use of this to distinguish themselves from each other. TASK can be cast to an unsigned int. The test file packaged with ARTK demonstrates another strategy for distinguishing between tasks with the same root function.
`void ARTK_Sleep( unsigned int ticks)`	Sleep for the indicated number of timer ticks. This allows other tasks to run during that time. The default timer tick is 10 msec, but may be changed by calling ARTK_SetOptions() from the user's Setup() function.
`void ARTK_Yield()`	Explicitly yield the processor. This can be used to force tasks of equal priority to share processor cycles earlier than they might otherwise.
`SEMAPHORE ARTK_CreateSema( int initial_count = 0)`	Create a semaphore with the initial count specified (defaults to 0).
`void ARTK_WaitSema( SEMAPHORE sem)`	Wait on the passed semaphore.
`void ARTK_SignalSema( SEMAPHORE sem)`	Signal the passed semaphore.
`int ARTK_WaitSema( SEMAPHORE semaphore, unsigned int timeout)`	Wait on a semaphore with a timeout specified as a number of ticks. This returns -1 if the call timed out, and 0 if the semaphore was acquired.
`void ARTK_TerminateMultitasking()`	ARTK will terminate when all tasks complete, or the user can terminate early by calling this function.

3 ARTK demo programme

The ARTK distribution includes the demo sketch in Listing 1, which illustrates the following:

- preemption of a lower priority task by a higher priority task
- waiting on a semaphore, both indefinitely and with a timeout
- signalling a semaphore from both a task and an ISR
- two methods for distinguishing between tasks with the same root function
- task sleeping
- stack size checking
- how the processor is shared between tasks of equal priority (not by timeslicing)
- measurement of latency between an ISR and a task
- estimating available RAM memory.

The serial channel output of the ARTK demo sketch is shown in Listing 2, followed by a step-by-step discussion of the multitasking events that lead to that output.

At start-up, ARTK prints a two line message identifying the release and the license. It then calls the user's Setup() function, which in this case modifies the default memory model for a large memory space, creates a semaphore, sets up some Arduino I/O pins, installs an ISR, prints a hello message with memory status, creates some tasks, and prints another memory status just before returning. From the memory status messages it can be seen that each task consumes 18 bytes of memory in addition to its stack space. Note that during a context switch, ARTK stores the stack pointer for a task in a task descriptor table and the rest of the processor state is stored on the task's stack.

Listing 1 ARTK demo sketch

```
// ARTK demo program for MEGA
// After uploading, open the Serial Monitor and press the processor reset button

#include <ARTK.h>

// these pins and interrupt numbers should work for UNO or MEGA
#define LED   13          // this pin has an LED on all Arduino boards
#define INTPIN 2          // we'll use this pin to trigger an interrupt
#define INTNUM 0          // this is the corresponding interrupt number

void producer(), consumer(), sleeper(), printer(), recurser(), myisr() ;
SEMAPHORE sem ;
long itime ;

// This will be called by ARTK once, before the scheduler is started. Use it as a
// replacement for the Arduino setup() routine.
void Setup()
{
  // changing to large model, leaving default sleep timer interval (10 msec)
  // changing the memory model MUST be done before creating any tasks
  ARTK_SetOptions(1,-1) ;

  // if an ISR that signals a semaphore is installed here, then the semaphore
  // should also be created here in case the ISR fires right away
  sem = ARTK_CreateSema(0) ;

  pinMode(LED, OUTPUT);                    // configure an output pin for the LED
  pinMode(INTPIN, OUTPUT);                 // we'll trigger an interrupt with a high
  digitalWrite(INTPIN, HIGH);              // to low transition on an output pin
  attachInterrupt(INTNUM, myisr, FALLING) ;

  Printf("Hello from Setup (%u avail)\n", ARTK_EstAvailRam() ) ;

  // create several tasks with varying priority and a default stack size of 256
  ARTK_CreateTask(sleeper, 4) ;
  ARTK_CreateTask(consumer, 3) ;
  ARTK_CreateTask(producer, 2) ;
  ARTK_CreateTask(printer, 1) ;
  ARTK_CreateTask(printer, 1) ;
  ARTK_CreateTask(recurser, 5) ;

  Printf("Setup returning (%u avail)\n", ARTK_EstAvailRam() ) ;
}

// this task demonstrates waiting on a semaphore
void consumer()
{
  int result = -1 ;
```

Listing 1 ARTK demo sketch (continued)

```
  Printf("consumer waiting on event\n") ;
  ARTK_WaitSema(sem) ;
  Printf("consumer received first event\n") ;

  Printf("consumer entering timed wait for second event\n") ;
  result = ARTK_WaitSema(sem, 5) ;
  while (result == -1)
  {
    Printf("consumer wait for second event timed out, trying again\n") ;
    result = ARTK_WaitSema(sem, 20) ;
  }

  Printf("consumer received second event, interrupt was %ld usec ago\n", micros()-itime) ;
  Printf("consumer exiting\n") ;
}

// this task demonstrates signaling a semaphore (it also triggers the ISR before exiting)
void producer()
{
  Printf("producer signaling event\n") ;
  ARTK_SignalSema(sem) ;
  Printf("producer sleeping for 20 ticks\n") ;
  ARTK_Sleep(20) ;
  Printf("producer triggering the interrupt service routine\n") ;
  digitalWrite(INTPIN, LOW) ;
  Printf("producer exiting\n") ;
}

// this ISR also signals the producer/consumer semaphore
void myisr()
{
  itime = micros() ;
  ARTK_SignalSema(sem) ;
}

// this task demonstrates sleeping, and is high enough priority to preempt
// other tasks in this demo, as seen in the output
void sleeper()
{
  Printf("Hello from sleeper\n") ;
  // give a message every 80 msec, then sleep - blink the LED too
  for (int i=0 ; i<10 ; i++)
  {
    digitalWrite(LED, ~digitalRead(LED)) ;
    Printf("Sleep %d\n", i) ;
    ARTK_Sleep(8) ;
  }
```

Listing 1 ARTK demo sketch (continued)

```
  Printf("sleeper exiting\n") ;
}

// two tasks are created with this root function and demonstrate two ways
// that they can distinguish themselves from each other
void printer()
{
  static int num = 1 ;
  int mynum = num++ ;
  for (int i=0 ; i<6 ; i++)
    Printf("Printer %d, id=%d, Value=%d\n", mynum, ARTK_MyId(), i) ;
  Printf("Printer %d exiting\n", mynum) ;
}

// This task calls a recursive function, printing the stack status each time
// The GNU GCC optimiser eliminates tail recursion, so the stack status is
// printed out AFTER the recursive call in order to see stack growth (as we climb
// out of the recursion)
void doAgain(int cnt)
{
  if (cnt==0) return ;
  doAgain(cnt-1) ;
  Printf("returning from doAgain %d, stack left = %d\n", cnt, ARTK_StackLeft()) ;
}
void recurser()
{
  Printf("Hello from recurser, stack left = %d\n", ARTK_StackLeft()) ;
  doAgain(5) ;
  Printf("recurser exiting\n") ;
}
```

Listing 2 ARTK demo sketch output

```
ARTK release 0.2
Paul Schimpf, 2012, GNU GPL
Hello from Setup (6527 avail)
Setup returning (4883 avail)
Start Tasking
Hello from recurser, stack left = 252
returning from doAgain 1, stack left = 227
returning from doAgain 2, stack left = 232
returning from doAgain 3, stack left = 237
returning from doAgain 4, stack left = 242
returning from doAgain 5, stack left = 247
recurser exiting
Hello from sleeper
Sleep 0
consumer waiting on event
producer signaling event
```

Listing 2 ARTK demo sketch output (continued)

```
consumer received first event
Sleep 1
consumer entering timed wait for second event
producer sleeping for 20 ticks
Sleep 2
Printer 1, id=2987, Value=0
consumer wait for second event timed out, trying again
Sleep 3
Printer 2, id=3261, Value=0
Printer 2, id=3261, Value=1
Printer 2, id=3261, Value=2
Sleep 4
Printer 1, id=2987, Value=1
producer triggering the interrupt service routine
Sleep 5
consumer received second event, interrupt was 68 usec ago
consumer exiting
Sleep 6
producer exiting
Printer 2, id=3261, Value=3
Printer 2, id=3261, Value=4
Printer 2, id=3261, Value=5
Sleep 7
Printer 1, id=2987, Value=2
Printer 1, id=2987, Value=3
Printer 1, id=2987, Value=4
Sleep 8
Printer 2 exiting
Printer 1, id=2987, Value=5
Printer 1 exiting
Sleep 9
sleeper exiting
All tasks done, exiting
```

Upon return from the user's Setup() function, ARTK prints a message that it is starting multitasking and performs a context switch to the highest priority task that is ready to run. In this case that is the recurser task, which performs a series of recursive function calls and prints the stack size remaining as it climbs out of the recursion. Note that the GNU GCC compiler used by the Arduino environment automatically eliminates tail recursion, so attempting to print the stack size before the recursive call here would result in no function calls and the stack would not change at all. From the stack size messages it can be seen that each call consumes 5 bytes on the stack: 3 bytes for a return address (large memory model) and 2 bytes for the int parameter that is passed.

The recurser task then exits, yielding control to the next highest priority task, which is the sleeper task. The sleeper task prints a hello message and then enters a loop wherein it blinks the Arduino LED, prints a 'Sleep <n>' message, where <n> is the loop count, and then sleeps for 80 msec. The first sleep call yields the processor to the next highest priority task, which is the consumer task.

The consumer task begins by entering an indefinite wait on the semaphore. This yields the processor to the producer task, which signals the semaphore. This results in an immediate context switch back to the consumer, which prints a message that it has received the first semaphore event. At this point in time the highest priority task, the sleeper, wakes up and preempts the consumer, printing a message and sleeping again. Control then returns to the consumer, which enters a timed wait on the semaphore. As a result, the processor is turned over to the producer, which continues execution at the return from signalling the semaphore. The producer then sleeps for 20 ticks.

At this point the sleeper wakes up, prints its second message, and sleeps again. The producer is also sleeping at this point, so control drops to priority level 1, and the first task in that ready queue is the first printer task. The first printer task gets as far as printing its first value when it is preempted by the consumer waking up from a timeout on the semaphore. The consumer then enters another timed wait, which again yields control down to priority level 1, but not before the sleeper wakes up again (Sleep 3). Note that priority level 1 contains two printer tasks, which will alternate their use of the processor when higher priority tasks are blocked. The second printer task manages to print 3 values before being preempted by the sleeper (Sleep 4). When the sleeper yields, control returns to the first printer task, which has time to print one value before being pre-empted by the producer task waking up from its sleep.

The producer task then triggers the interrupt with a write to the interrupt pin. Note that the Atmel processors used on Arduino allow an interrupt to be triggered by writing to a processor's interrupt pin, which is useful for generating interrupts from either hardware or software. Interrupts run in the context of whatever task they occur in, which in this case is the producer task. The ISR takes a time stamp and signals the producer/consumer semaphore. This causes a context switch to the consumer task, which is waiting on that semaphore and has a higher priority than the producer task. There is an intervening pre-emption by the sleeper task (Sleep 6) and the consumer reports acquisition of the event along with the delay between the execution of the ISR and its acquisition of the semaphore (in this case 68 microseconds).

The consumer then exits, there is a pre-emption by the sleeper (Sleep 7) and control returns to the producer task, which exits. This yields control to priority level 1 again, and the two printer tasks take turns each time they are preempted by the sleeper, until both are done printing their values.

When both printer tasks complete, the processor is idle while the sleeper task is sleeping. The sleeper then prints its last value and exits. ARTK recognises that all tasks have completed, prints a message to that effect, and shuts down multitasking.

4 Conclusions

ARTK is a simple but effective real-time kernel, with a very small footprint, that can provide ease and clarity in the programming of concurrent activities in embedded systems powered by the Arduino.

References

Arduino (2012) [online] http://www.arduino.cc/ (accessed 13 June 2012).

Buhr, R.J.A. and Bailey, D.L. (1999) *An Introduction to Real-Time Systems, from Design to Networking with C/C++*, Prentice-Hall, New Jersey.

Duinos (2011) [online] http://robotgroup.com.ar/duinos/wiki/index.php?title=Main_Page (accessed 13 June 2012).

Sha, L. and Goodenough, J.B. (1990) 'Real-time scheduling theory and Ada', *Computer*, Vol. 23, No. 4, pp.53–62.

Tindell, K. (2000) 'Deadline monotonic analysis', *EE Times-India*, [online] http://www.eetindia.co.in/ART_8800505178_1800000_TA_4b544ed8.HTM (accessed 13 June 2012).