ArdOS RTOS implementation

Here we will show how to setup and work in one of many RTOS implementations – *ArdOS* (short from *The Arduino Operating System*). This RTOS implementation allows usage of the majority of services explained before on numerous Arduino platforms, including Arduino UNO.
ArdOS main features are:
• full compatibility with standard Arduino libraries, as well as with *Arduino IDE*;
• small, compact OS *kernel*;
• priority scheduler for multitasking support in hard RT applications;
• maximal task number is 8;
• implementation of *sleep* function within a task which allows stopping current task during given time interval, while other tasks can be run during this time interval;
• binary and counting semaphores;
• mutex and conditional variables;
• FIFO and priority queues;
• configurability which allows removing non-needed segments of OS in order to reduce memory consumption.

Original version of ArdOS can be downloaded from the link below.

https://bitbucket.org/ctank/ardos-ide/wiki/Home

ArdOS structure

Operating system ArdOS is implemented as a set of files where each file contains specific OS functional unit. Those files are:

• *kernel.c* and *kernel.h* – files that implement the kernel of an operating system. This consists of functions for initialization and starting of OS, initialization of system time, interrupt service routine for system time measurement, functions for creating tasks, functions for switching tasks, assembly code for saving and restoring the context of a task etc.;
• *task.c* – file that contains additional functions for managing the queue which stores tasks;
• *profiler.c* and *profiler.h* – files that implement functions for measurement of task execution duration, as well as storing them within EEPROM memory;
• *sema.c* and *sema.h* – library which contains functions for creating and management of semaphores;
• *mutex.c* and *mutex.h* – library which implements functions for creating and management of mutexes and conditional variables;
• *queue.c* and *queue.h* – library which implements functions for creating and management of FIFO and priority queues.
Above mentioned files contain numerous functions, variables and constants which are required for correct OS functioning. However, in order to use it, it is not needed to know and understand each of them. Hence, we will only describe functions which user has to know and use properly. In order to enable *ArdOS*, user has to go through the following steps:
1. *Include required libraries* – Standard library for working with ArdOS is *kernel.h*. If it is needed to use additional services as a part of the project, it is mandatory to include libraries implementing them, as well.
2. *OS initialization* – this assumes calling function *OSInit* which specifies the number of the tasks which OS will support.
3. *Create service* – If it is needed to use a service within a project, user must define it as a global variable and create it using a dedicated function.

4. *Create task* – It is required to define as many functions, implementing the code to be executed by a task, as there are tasks specified during OS initialization. Also, it is mandatory to attach defined functions to corresponding tasks.

5. *Starting OS* – As the last step, OS can be started by using function *OSRun()*.

Functions for initializing and starting OS

Two main functions are to be used in order to start OS: *OSInit()* and *OSRun()*. Both of them are defined within the header file *kernel.h*.

```
void OSInit ( uint8_t numTasks );
```

Description: Function initializes OS and specifies the number of tasks to be executed by it.
Parameters:
• numTasks – number of tasks which is to be executed concurrently; maximum value of this parameter is 8, which limits the number of tasks in a system to 8.
Return value:
None.

```
void OSRun ();
```

Description: Function which starts OS.
Parameters:
None.
Return value:
None.

Functions for creating tasks

Functions used during creation of the task are *OSSetStackSize()* and *OSCreateTask()*. Both functions are defined within header file *kernel.h*.

```
void OSSetStackSize ( uint8_t stackSize );
```

Description: Function adjusts the size of the stack which will be assigned to each task.
Parameters:
• stackSize – number which represents the size of the stack where each stack element is of type uint32_t, default value of the parameter is 50, which means that stack will occupy 200 bytes.
Return value:
None.

```
uint16_t OSCreateTask ( uint8_t prio , void (* fptr )( void *) , void* param );
```

Description: Function is used for creating tasks. It will register new task, adjust the task parameters like priority and stack. Then, it will assign to task the function which represents task and define its parameters.
Parameters:
• prio – number which represents the priority of the task created; value can be in the range 0 to numTasks – 1, where numTasks is the number of tasks specified during OS initialization; lower value stands for higher priority, meaning that task with priority 0 has the highest priority in the system, while the task with priority numTasks-1 has the lowest. In ArdOS each task has to have different priority level.
• fptr – pointer to the function which will be invoked by the created task
• param – pointer to the set of arguments which needs to be passed to the function when the task is started
Return value:

Error code:
• OS_NO_ERR – function returns with no error
• OS_ERR_DUP_PRIO – given priority is already used when previous task is created
• OS_ERR_BAD_PRIO – given priority is not in proper priority range from 0 to numTasks-1

Functions for task state control

During OS runtime, tasks are executed in so-called pseudo-parallel manner, where parallelism is obtained by alternating task execution. Such OS operation requires functions that perform task swapping and they can be invoked either explicitly as a part of some other task or implicitly by some OS function (which is a typical case). For example, invoking function *OSSleep()* will put currently running task to „sleep“ and select other ready task to move into running state (i.e. to be executed by OS). Depending on the way this task switch is done, there are two functions: *OSSwap()* and *OSPrioSwap()*.

```
void OSSwap ();
```
Desciption: Function allows task to discard microcontroller. In case when multiple tasks are in ready state, the one with the highest priority will be executed next. If no other task is ready, control is returned to the task that called this function.
Parameters:
None.
Return value:
None.

```
void OSPrioSwap ();
```
Desciption: Function which allows a task to discard microcontroller and stop its execution. In this case, a task with the highest priority which is ready will be executed next, but provided that its priority is higher than the priority of currently running task. If no such task exists, control will be returned to the task that invoked the function.
Parameters:
None.
Return value:
None.

Time management functions

Operating system ArdOS supports two such functions: *OSSleep()* and *OSticks()*.

```
void OSSleep ( uint32_t ms);
```
Description: Function which blocks or "sleeps" the task not less than for a time interval specified by a function argument *ms*. When specified time interval elapses, the same task will continue to execute only if it has the highest priority. This means that the tasks with higher priority will have more accurate sleeping time.
Parameters:
• ms – number of milliseconds during which the task will be blocked; this parameter is an unsigned 32-bit value
Return value:
None.

```
uint32_t OSticks ();
```
Description: Function which returns number of milliseconds elapsed since OS started running.
Parameters:
None.
Return value:

Number of milliseconds passed from the moment when OS was started by function *OSRun()*. Number of milliseconds is an unsigned 32-bit value.

Functions for managing semaphores

Semaphores are services which are controlled by one of three functions: *OSCreateSema()*, *OSTakeSema()* and *OSGiveSema()*. Since multiple semaphores can be used within the application, it is necessary to create objects of type *OSSema* as a global variables, after which they need to be initialized.

```
void OSCreateSema ( OSSema *sema , uint16_t initVal , uint8_t isBinary );
```

Description: Function which initializes the semaphore object, based on give arguments.
Parameters:
• sema – pointer to semaphore which needs to be initialized
• initVal –value used for initializing semaphore. If semaphore is a binary semaphore, this value can be 0 or 1, if semaphore is a counting semaphore any non-negative value can be used here
• isBinary – parameter which determine if the semaphore is configured as a binary or counting semaphore; if parameter is a positive number, semaphore is used as a binary semaphore, otherwise a counting semaphore will be created
Return value:
None.

```
void OSTakeSema ( OSSema * sema );
```

Description: Function which tries to take the semaphore. If semaphore has a value different from 0, this will be completed successfully and value of the semaphore will be decremented. Otherwise, the task will be blocked.
Parameters:
• sema – pointer to the semaphore which is to be taken
Return value:
None.

```
void OSGiveSema ( OSSema * sema );
```

Description: Function which gives the semaphore and unblocks the task of the highest priority which is waiting for it, if it exists. Otherwise, it will increment the value of the semaphore.
Parameters:
• sema – pointer to the semaphore which is to be given
Return value:
None.

Functions for managing mutexes

By their concept, functions which work with mutex objects are similar to the ones used for semaphores. OS supports several mutex objects at the same time, so it is important to create and initialize all required objects of type *OSMutex* as global variables. Functions that work with mutexes are *OSCreateMutex()*, *OSTakeMutex()* and *OSGiveMutex()*.

```
void OSCreateMutex ( OSMutex * mutex );
```

Description: Function which initializes mutex object.
Parameters:
• mutex – pointer to mutex object that is to be initialized
Return value:
None.

```
void OSTakeMutex ( OSMutex * mutex );
```

Description: Function which tries to acquire the mutex. If mutex if free, current task will continue executing. Otherwise, the task will be blocked until the mutex is released by the other task.
Parameters:
• mutex – pointer to the mutex which is to be acquired
Return value:
None.

```
void OSGiveMutex ( OSMutex * mutex );
```
Description: Function which releases mutex object. If there are several tasks waiting for this mutex to be released, the highest priority task will be unblocked and ready for execution. If the task which releases the mutex has the highest priority compared to all the other tasks waiting for a mutex, this task will continue execution after the mutex is released.
Parameters:
• mutex – pointer to mutex which is to be released
Return value:
None.

Application examples

Here we will show few *ArdOS* application examples. In order to run these examples, only requirement is to have an Arduino board connected to a host computer using serial communication interface.

Task 1: Implement application which allows simultaneous blink of a LED with frequency of 2Hz (twice per second) and a system clock printout using Serial object each 500ms.

Solution: Since we need simultaneous execution of two functionalities, we will use two tasks. First task will be implemented within function *task1*, which will send a message containing system time each 500ms. Second task which provide that LED blinks with frequency of 1Hz and function assigned to this task will be *task2*. Parameter of this function could be pause between on and off, which, in this case, will be set to 250ms to provide 2 blinks per second. This time interval is also sent as a parameter to function when the task is created. Solution to the task is given below.

```
#include <kernel.h>
#define NUMBER_TASKS 2
// First task writes OSticks to the serial port
void task1(void *p)
{
	char buffer [16];
	// Sending message using serial port
	while (1)
	{
		sprintf (buffer , "Task1: %lu", OSticks());
		Serial.println(buffer);
		OSSleep (500);
	}
}
// Second task blinks LED with pause parameter p
void task2(void *p)
{
  unsigned int pause=(unsigned int) p;
  while(1)
  {
```

```
    digitalWrite(13, HIGH);
    OSSleep(pause);
    digitalWrite(13, LOW);
    OSSleep(pause);
  }
}

void setup()
{
  OSInit(NUM_TASKS);

  Serial.begin(115200);
  pinMode(13, OUTPUT);

  OSCreateTask(0, task1, NULL);
  OSCreateTask(1, task2, (void *) 250);

  OSRun();
}

void loop()
{
  // Empty
}
```

Task 2: Implement application which waits for character '*S'* sent over serial port. When character is received, it will start blinking LED with frequency of 1Hz. Additionally, system should be able to receive messages even during the LED blinks.

Solution: Since we have two synchronous functionality to implement, we will need two tasks. First task is in charge of accepting characters over serial port, check the validity and signaling to the other task that blinking start is required. Second task will be implemented through the function *task2*, with a parameter representing duration of a single blink. Task 2 will initially try to take semaphore and will be blocked since semaphore is not available. Semaphore will become available when task 1 detects received character 'S' after which it will call *OSGiveSema* function (making semaphore available). In this case, we are using a binary semaphore.

```
#include <kernel.h>
#include <sema.h>

#define NUM_TASKS    2

OSSema sem; //semaphore definition

// START task
void task1(void *p)
{
  unsigned char flag=0, start;

  while(1)
  {
    Serial.print("Type character S to start LED pulsing !\r\n");

    while(Serial.available() == 0)
         OSSleep(10) ;

    OSSleep(100);

    start = Serial.read();
```

```
    flag = 0;

    if(start != 'S')
      flag = 1;

    if(flag == 0)
    {
      Serial.print("SUCCESS !\r\n");
      OSGiveSema(&sem); // Give semaphore
    }
    else{
      Serial.print("FAIL !\r\n");
    }
  }
}

void task2(void *p)
{
  OSTakeSema(&sem); // Taking semaphore
  unsigned int pause=(unsigned int) p;

  while(1)
  {
    digitalWrite(13, HIGH);
    OSSleep(pause);
    digitalWrite(13, LOW);
    OSSleep(pause);
  }
}

void setup()
{
  OSInit(NUM_TASKS);
  OSCreateSema(&sem , 0, 0); // Initialize semaphore
  Serial.begin(115200);
  pinMode(13, OUTPUT);
  Serial.println("Test !\r\n");
  OSCreateTask(0, task1, NULL);
  OSCreateTask(1, task2, (void *) 500);

  OSRun();
}

void loop()
{
  // Empty
}
```

Task 3: Implement application which has 3 tasks. Tasks 1 and 2 send three messages each (total of 6 messages for one cycle) in the loop, sleeping after each message 200ms. Task 3 blinks LED with frequency 1Hz. Observe the order of sending messages.

Task 4: Modify previous application in order to provide that each task sends all three messages before messages from the other task are sent. Order of messages should be:

Task1: 0
Task1: 1
Task1: 2
Task2: 0

Task2: 1
Task2: 2
Task1: 0
Task1: 1
Task1: 2
Task2: 0
Task2: 1
Task2: 2

Solution: Since we need to prevent other message-sending-task (i.e. task2) from "jumping in" and sending messages before all three messages from task1 are sent, we will use mutex for providing **mutual exclusion** of serial port usage.

```
#include <kernel.h>
#include <mutex.h>

// Constant which allows mutex usage (all mutex usages will be wrapped by this
//constant
#define USE_MUTEX
OSMutex mutex ; // Creating mutex object

void task1 (void * p)
{
  int i ;
  char msg [64];
  while (1)
  {
    #ifdef USE_MUTEX
      OSTakeMutex (&mutex) ; // Take mutex
    #endif
    // Printout message
    for ( i = 0; i < 3; i++)
    {
      sprintf( msg ,"Task1 : %d \r\n", i ) ;
      Serial.println(msg) ;
      OSSleep(200) ;
    }
    #ifdef USE_MUTEX
      OSGiveMutex (&mutex) ; // Give mutex so other task can take it
    #endif
  }
}

void task2 (void * p)
{
  int i ;
  char msg [64];
  while (1)
  {
    # ifdef USE_MUTEX
    OSTakeMutex (&mutex) ; // Take mutex
    # endif
    // Message printout
    for (i = 0; i < 3; i++)
    {
      sprintf (msg , "Task2 : %d \r\n", i) ;
      Serial.println(msg) ;
      OSSleep(200);
```

```
    }
    #ifdef USE_MUTEX
      OSGiveMutex (&mutex) ; // Give mutex so other task can take it
    #endif
  }
}


void task3 (void * p)
{
  //Blink led with frequency 1Hz
  while(1)
  {
    digitalWrite(13, HIGH);
    OSSleep(500);
    digitalWrite(13, LOW);
    OSSleep(500);
  }
}

void setup()
{
  //Initialize pin as output and serial port
  pinMode(13, OUTPUT);
  Serial.begin(115200);
  //Initialize OS with 4 tasks that we will use
  OSInit(3);
  // Initialize mutex
  OSCreateMutex (&mutex) ;
  // Initialize tasks
  OSCreateTask (0, task1, 0) ;
  OSCreateTask (1, task2, 0) ;
  OSCreateTask (2, task3, 0) ;
  OSRun() ; // Start OS
}

void loop(){
  //nothing to do here
}
```

Task 5: Repeat previous task but add additional task which sends messages in the same manner as task1 and task2. So, task1, task2 and task3 are the tasks sending messages guarded by mutex object, while task 4 is the task that blinks LED with frequency 1Hz. What can you observe from the output?

Additional tasks for practicing

Task 6: Implement application which waits for characters '*S*' or '*R*' sent over serial port. When character '*S*' is received, it will start blinking LED with frequency of 1Hz. When character '*R*' is received, it will stop with LED blinking.

Task 7: Implement application which waits for digits 1-9 sent as characters over the serial port. The digit received over serial port represents a number of LED blinks which will follow. For example, if character '3' is received, resulting behavior will be 3 LED blinks. Program should continue waiting for messages even after message is received. **Hint: Try using the counting semaphore initialized according to a digit value of a received character.**

Task 8: Implement application for LED dimming with two tasks. One task performs LED toggling with frequency of 50Hz (*on_time* 10ms and *off_time* 10ms). Other task waits for characters '*U*' or '*D*' sent over serial port. When character '*U*' is received, *on_time* is increased and *off_time* is decreased by 1, and opposite when character 'D' is received. In both cases, the sum *on_time* + *off_time* should remain the same in order not to change toggling frequency. **Note: don't allow either *on_time* or *off_time* to drop to 0.**