ArdOS – The Arduino Operating System
Reference Guide

Contents

1. Introduction

ArdOS is an advanced operating system for the Arduino series of microcontroller boards. It features full compatibility with the Arduino library and IDE, and is fully configurable to offer you flexibility and space-saving.

This document provides you with details on the ArdOS application programming interface. For details on how to set up ArdOS and for a quick tutorial, please look up the Quick Start Guide.

2. Error Handling

In the event of serious error, ArdOS lights up the "L" LED on the Arduino board (corresponding to digital pin 13) and returns the error code via the serial port connected to the PC, which can be read using the Serial Monitor. The table below shows the possible errors:

Error Code	Name	Description
1	OS_ERR_MAX_PROCS	Attempting to create more tasks than declared when OSInit was called.
2	OS_ERR_BAD_PRIO	Priority number out of range. Priorities should range from 0 to *n-1*, where *n* is the number of tasks declared when OSInit was called.
3	OS_ERR_DUP_PRIO	Duplicate priority number. Tasks must have unique priority numbers between 0 and *n-1*.
4	OS_ERR_NOMEM	Out of RAM, usually as a result of attempting to allocate a new stack that is too large.

3. Initialization and Startup

These functions initialize ArdOS and start it running:

3.1 Initializing and Starting ArdOS

Function Definition	Parameters	Description
`void OSInit(unsigned char numTasks)`	numTasks: Number of tasks you intend to create.	Initializes ArdOS. This should be the first function called before anything else.
`void OSRun()`	-	Starts up the operating system, which will begin executing the highest priority task.

4. Task Creation

Tasks are the basic units in ArdOS, and they roughly correspond to the various items that an application has to take care of. A robot for example might have one task that monitors readings from a sonar sensor to gauge distances to obstacle, another that displays information on an LCD on the robot, a third task that manipulates the motors, and finally a fourth task that determines the high level behaviors of the robot based on the sensors.

4.1 Setting the Task Stack Size

Function Definition	Parameters	Description
`void OSSetStackSize(unsigned char stackSize)`	stackSize: Stack size in longs (4-byte). Default value is 50.	Sets the stack size of the next task to be created using OSCreateTask.

4.2 Creating a New Task

Function Definition	Parameters	Description
`unsigned int OSCreateTask(unsigned char prio, void (*fptr)(void *), void *param)`	prio: Priority number for the current task. Ranges between 0 and numTasks-1. Smaller priority numbers have higher priorities (i.e. 0 is highest, numTasks-1 is lowest). fptr: Pointer to the task function. param: Argument to be passed to the task on startup.	Registers a new task. The task will have a stack size determined by OSSetStackSize (default stack size is 50 32-bit words).

5. Task Management

Task management functions let a task go to sleep for a specific amount of time, or allows a task to pass control over to another task.

5.1 Putting a Task to Sleep

Function Definition	Parameters	Description
`void OSSleep(unsigned long ms)`	ms: Number of milliseconds to sleep.	This puts a task to sleep for at least the specified number of milliseconds. When the number of milliseconds has passed, a task will restart only if it currently has the highest priority. This means that tasks with higher priorities will have more accurate sleep times.

5.2 Handing Control to another Task

Function Definition	Parameters	Description
`void OSSwap()`	-	Allows a task to relinquish control of the microcontroller. If no other tasks are ready to run, control is handed back to the calling task.
`void OSPrioSwap()`	-	Similar to OSSwap but hands control over only if there is a HIGHER PRIORITY task that is currently ready to run. If only lower priority tasks are ready to run control is handed back to the calling task.
`void OSSwapFromISR()`	-	Hands control over from within an interrupt handler. Equiavlent to OSSwap.
`void OSPrioSwapfromISR()`	-	Hands control over from within an interrupt handler. Equivalent to OSPrioSwap.

5.3 Miscellaneous Kernel Functions

Function Definition	Parameters	Description
`void OSticks()`	-	Returns the number of milliseconds (ticks) since OSRun() was called.

6. Semaphores

Semaphores are useful for coordinating tasks. A counting semaphore is a semaphore that can take on a value that is any positive integer including zero. It could be used, for example, to track the number of slots left in a buffer, suspending a task when there are no more slots left and resuming it when slots are freed up.

A binary semaphore, as its name suggests, takes on a value of either 0 or 1. It can be used to prevent a task from running until a specific event occurs. An example usage is given in the ISRDemo sketch where a task is suspended until a push button is pressed.

6.1 Declaring and Initializing Semaphores

Semaphores are declared using the OSSema structure. For example:

```
OSSema sema; // Create a semaphore called sema.
```

Once a semaphore has been declared you can initialize it using the OSCreateSema call:

Function Definition	Parameters	Description
`void OSCreateSema(OSSema *sema, unsigned int initVal, unsigned char isBinary)`	sema: The semaphore to initialize. initVal: Starting value for the semaphore. isBinary: Putting a non-zero value here creates a binary semaphore, putting a 0 here creates a counting semaphore.	Sets the initial value (0 or 1 for binary semaphore, any positive integer including 0 for a counting semaphore) and type of the semaphore.

6.2 Taking and Giving Semaphores

If a task takes a semaphore that is currently 0, the task will block and control will be handed over to another task. If the task takes a semaphore that is not zero, the take call returns immediately and the calling task resumes.

If a task gives a semaphore, if there are any tasks blocking on the semaphore, the task with the highest priority is unblocked and becomes ready to run. If the unblocked task has a higher priority than the calling task, the calling task is pre-empted and the unblocked task will run.

If there are no tasks blocking on the semaphore, the semaphore is simply incremented (or set to 1 in the case of binary semaphores).

Function Definition	Parameters	Description
`void OSTakeSema(OSSema *sema)`	sema: The semaphore to take.	Returns if sema is non-zero, otherwise blocks.
`void OSGiveSema(OSSema *sema)`	sema: The semaphore to give.	Unblocks the highest priority task waiting on sema if any. Otherwise increments sema (or sets it to 1).

7. Queues

While it is possible to pass messages through global variables and semaphores, queues provide a safer and more convenient method to pass messages between tasks. To simplify queue management ArdOS assumes that several tasks might write to a queue, but at most one task would read from it. ArdOS provides support for conventional first-in-first-out (FIFO) queues and prioritized message queues.

A queue in ArdOS consists of two parts: A buffer, and a queue data structure. Both must be provided to ArdOS when initializing a queue.

7.1 First-in-first-out (FIFO) queues.

In FIFO queues, as the name suggests, messages are read off the queue in the same order that it was written in. FIFO queues are the most common form of queues and the fastest. As such FIFO queues are always recommended unless you require prioritized messaging.

7.1.1 Declaring and Initializing FIFO Queues

A FIFO queue consists of an integer array and a queue data structure of type OSQueue.

```
#define BUFLEN 8
int buffer[BUFLEN];
OSQueue queue;
```

Call the OSCreateQueue function to initialize the queue:

Function Definition	Parameters	Description
`void OSCreateQueue(int *buffer, unsigned char length, OSQueue *queue)`	buffer: The integer buffer you declared to store the messages. length: The length of the integer buffer. queue: Pointer to the queue structure.	Initializes a FIFO queue.

7.1.2 Reading and Writing FIFO Queues

Two functions are provided to access FIFO queues. OSEnqueue writes a new message to the queue, unblocking any tasks that might be blocked trying to read an empty queue. If the queue is full the latest written message is lost. It is therefore recommended that the task reading from the queue should be given a higher priority than the tasks writing to the queue.

OSDequeue reads previously written messages and blocks if there are no messages to be read.

Function Definition	Parameters	Description
`void OSEnqueue(int data, OSQueue *queue)`	data: Message to be written to the queue. queue: Queue to write to. Must be previously initialized with OSCreateQueue.	Writes data onto the queue. If the queue is full data will be lost. To minimize this tasks that write to a queue should be given lower priority than tasks that read from a queue. If any call is blocked on OSDequeue on "queue", OSEnqueue causes the task to become unblocked and moved to the ready state.
`int OSDequeue(OSQueue *queue)`	queue: Queue to read from.	Returns message at the head of the queue. Messages are read off in the same order that they were written, i.e. in first-in-first-out (FIFO) order. This call blocks if there is nothing to read. **At most one task should call OSDequeue.**

7.2 Prioritized Message Queues

In prioritized message queues messages are assigned priorities when enqueued, and are dequeued in order of priority. There are additional overheads involved in keeping the queue in priority order so it is recommended that you use FIFO queues unless you require prioritized messaging.

7.2.1 Declaring and Initializing Priority Queues

A priority queue consists of an array of type TPrioNode and a queue data structure of type OSQueue.

```
#define BUFLEN    8
TPrioNode buffer[BUFLEN];
OSQueue queue;
```

Call the OSCreatePrioQueue function to initialize the queue:

Function Definition	Parameters	Description
`void OSCreatePrioQueue(TPrioNode *buffer, unsigned char length, OSQueue *queue)`	buffer: The TPrioNode buffer you declared to store the messages. length: The length of the integer buffer. queue: Pointer to the queue structure.	Initializes a priority queue.

7.2.2 Writing and Reading Priority Queues

Two functions are provided to access FIFO queues. OSEnqueue writes a new message to the queue, unblocking any tasks that might be blocked trying to read an empty queue. If the queue is full the latest written message is lost. It is therefore recommended that the task reading from the queue should be given a higher priority than the tasks writing to the queue.

OSDequeue reads previously written messages and blocks if there are no messages to be read.

Function Definition	Parameters	Description
`void OSPrioEnqueue(int data, unsigned char prio, OSQueue *queue)`	data: Message to be written to the queue. prio: Message priority. Higher priority messages have smaller priority numbers and vice versa. queue: Queue to write to. Must be previously initialized with OSCreateQueue.	Writes data onto the queue. If the queue is full data will be lost. To minimize this tasks that write to a queue should be given lower priority than tasks that read from a queue. Messages are stored in the queue in order of priority, with highest priority messages put in front of the queue. If any call is blocked on OSDequeue on "queue", OSEnqueue causes the task to become unblocked and moved to the ready state.
`int OSDequeue(OSQueue *queue)`	queue: Queue to read from.	Returns message at the head of the queue. Messages are read off in order of priority. This call blocks if there is nothing to read. **At most one task should call OSDequeue.**

8. Mutex Locks and Conditional Variables

A mutex lock (short for "mutual exclusion lock") can be used to prevent more than one task from executing a block of code or accessing a shared variable at the same time. They are similar to binary semaphores are initialized to 1, and they can be used together with conditional variables.

A conditional variable is a variable that a task can wait on, and other tasks can use this variable to signal a waiting task.

8.1 Mutex Locks

8.1.1 Declaring and Initializing Mutex Locks

Mutex locks are declared as global variables of type OSMutex:

```
OSMutex mutex;
```

A mutex must be initialized using OSCreateMutex:

Function Definition	Parameters	Description
`void OSCreateMutex(OSMutex *mutex)`	mutex: Mutex lock to be initialized.	Initializes the mutex lock.

8.1.2 Taking and Releasing Mutex Locks

Before entering a shared function or accessing a shared variable, a task should first attempt to take a mutex lock. If the lock is free the task can call the shared function or access the shared variable. If the lock has been taken by another task the calling task is blocked until the lock has been released by the other task.

Function Definition	Parameters	Description
`void OSTakeMutex(OSMutex *mutex)`	mutex: Mutex lock to be taken.	If the lock is free this function returns immediately. If the lock has been taken by another task this call will block until the lock is released.
`void OSGiveMutex(OSMutex *mutex)`	mutex: Mutex lock to be released.	This call releases a mutex lock. If any task is blocking on the mutex lock, the highest priority task is unblocked and becomes ready to run. The task calling OSGiveMutex will be pre-empted if the unblocked task has a higher priority.

8.2 Conditional Variables

A conditional variable is a special variable that can be used to tasks can wait on and be signaled. If a "signal" call is issued before a task actually "waits", the "signal" is often lost and the tasks eventually deadlock. To prevent this conditional variables are called within "mutual exclusions" enforced by mutex locks.

8.2.1 Declaring and Initializing Conditional Variables

A conditional variable is a global variable that is declared with type OSCond.

```
OSCond cond;
```

Conditional variables must be initialized using OSCreateConditional before being used:

Function Definition	Parameters	Description
`void OSCreateConditional(OSCond *cond)`	cond: The conditional variable to be initialized.	The conditional variable cond is initialized and ready to be used with OSWait and OSSignal.

8.2.2 Waiting and Signaling on Conditional Variables

To ensure correct operation conditional variables must be called within "mutual exclusions" enforced by mutex locks. The OSWait function releases the specified mutex lock before being put to sleep. When another task wakes it with a signal call, the sleeping task automatically re-acquires the mutex lock and must release it at the end.

Function Definition	Parameters	Description
`void OSWait(OSCond *cond, OSMutex *mutex)`	cond: The conditional variable to wait on. mutex: The mutex lock that is securing the code that contains the OSWait call.	"mutex" is released and the task is put to sleep. **At most one task can wait on a single conditional variable.**
`void OSSignal(OSCond *cond)`	cond: The conditional variable to signal.	The task (if any) that is waiting on cond will be unblocked and become ready to run. The task will be run when it becomes the highest priority ready task. When the signaled task resumes it will automatically acquire the mutex lock again. All tasks included signaled tasks must therefore remember to release the mutex lock before exit.

9. Advanced Configuration

To help conserve memory parts of ArdOS can be switched off by setting compiler variables in the configuration section of kernel.h. This section begins with the header:

```
/*
=============================================================================

      CONFIGURATION SECTION:

      Use this section to configure various parts of ArdOS.
=============================================================================
*/
```

The table below describes what each option does:

Option	Possible Values	Default Value	Description
OSCPU_TYPE	AT168 AT328 AT1280 AT2560	Automatically set.	Microcontroller type to compile for. This option is automatically set by the Arduino IDE.
OSMAX_TASKS	Any positive integer	8	The absolute maximum number of tasks that your system can support.
OS_PREEMPTIVE	0 or 1	1	Switches on pre-emption. When a higher priority task becomes ready to run, the currently running task is automatically pre-empted. Set to 0 to turn on co-operative mode. In this mode tasks continue to run until they call OSSwap or OSPrioSwap. See Section 4.2 for details on these calls.
OSSTACK_SIZE	0 to 255	50	Sets the stack size (in 32-bit words) of the next task to be created using OSCreateTask. If the microcontroller frequently reboots unexpectedly, try setting this to a higher value. You can set this to a lower value to save memory. ArdOS returns error code 4 (OS_ERR_NOMEM) if there is insufficient RAM to allocate the stacks.
OSUSE_SLEEP	0 or 1	1	If 1, generates code to support the OSSleep function.
OSUSE_SEMA	0 or 1	1	If 1, generates code to support semaphores.
OSUSE_QUEUES	0 or 1	1	If 1, generates code to support FIFO message queues.
OSUSE_PRIOQUEUES	0 or 1	1	If 1, generates code to support prioritized message queues.
OSUSE_MUTEXES	0 or 1	1	if 1, generates code to support mutex locks.
OSUSE_CONDITIONALS	0 or 1	1	If 1, generates code to support conditional variables.