[](https://raw.githubusercontent.com/ivanseidel/ArduinoThread/master/extras/ArduinoThread.png)

Arduino does not support "REAL" parallel tasks (aka Threads), but we can make use of this Library to improve our code, and easily schedule tasks with fixed (or variable) time between runs.

This Library helps to maintain organized and to facilitate the use of multiple tasks. We can use Timers Interrupts, and make it really powerfull, running "pseudo-background" tasks under the rug.

For example, I personaly use it for all my projects, and put all sensor aquisition and filtering inside it, leaving the main loop, just for logic and "cool" part.

**ArduinoThreads is a library for managing the periodic execution of multiple tasks.**

Blinking an LED is often the very first thing an Arduino user learns. And this demonstrates that periodically performing one single task, like toggling the LED state, is really easy. However, one may quickly discover that managing multiple periodic tasks is not so simple if the tasks have different execution periods.

ArduinoThreads is designed to simplify programs that need to perform multiple periodic tasks. The user defines a Thread object for each of those tasks, then lets the library manage their scheduled execution.

It should be noted that these are not “threads” in the real computer-science meaning of the term: tasks are implemented as functions that are periodically run to completion. On the one hand, this means that the only way a task can “yield” the CPU is by returning to the caller, and it is thus inadvisable to delay() or do long waits inside a task. On the other hand, this makes ArduinoThreads memory friendly, as no stack needs to be allocated per task.

**Installation**

1. "Download":<https://github.com/ivanseidel/ArduinoThread/archive/master.zip> the Master branch from gitHub.
2. Unzip and modify the Folder name to "ArduinoThread" (Remove the '-master')
3. Paste the modified folder on your Library folder (On your Libraries folder inside Sketchbooks or Arduino software).
4. Restart Arduino IDE

**If you are here, because another Library requires this class, just don't waste time reading bellow. Install and ready.**

**Getting Started**

There are many examples showing many ways to use it. Here, we will explain Class itself, what it does and "how" it does.

There are basicaly, three Classes included in this Library: Thread, ThreadController and StaticThreadController (both controllers inherit from Thread).

* Thread class: This is the basic class, witch contains methods to set and run callbacks, check if the Thread should be runned, and also creates a unique ThreadID on the instantiation.
* ThreadController class: Responsable for "holding" multiple Threads. Can also be called as "a group of Threads", and is used to perform run in every Thread ONLY when needed.
* StaticThreadController class: Slighly faster and smaller version of ThreadController. It works similar to ThreadController, but once constructed it can't add or remove threads to run.
* The instantiation of a Thread class is very simple:

Thread myThread = Thread();

// or, if initializing a pointer

Thread\* myThread = new Thread();

* Setting up a thread is essential. You can configure many things:

myThread.enabled = true; // Default enabled value is true

myThread.setInterval(10); // Setts the wanted interval to be 10ms

/\*

This is useful for debugging

(Thread Name is disabled by default, to use less memory)

(Enable it by definint USE\_THREAD\_NAMES on 'Thread.h')

\*/

myThread.ThreadName = "myThread tag";

// This will set the callback of the Thread: "What should I run"?

myThread.onRun(callback\_function); // callback\_function is the name of the function

Ok, creating Threads are not so hard. But what do we do with them now?

* First, let's see how Threads should work, to understand what a ThreadController is and does

// First check if our Thread "should" be runned

if(myThread.shouldRun()){

// Yes, the Thread should be runned, let's run it

myThread.run();

}

Now that you got the idea, let's think a little bit: What if i have 3, 5, 100 Threads. Do I need to check EACH one?!?

* The answer is: NO. Create a ThreadController or StaticThreadController, and put all your boring-complex Threads inside it!

// Instantiate a new ThreadController

ThreadController controller = ThreadController();

// Now, put a bunch of Threads inside it, FEED it!

controller.add(&myThread); // Notice the & before the thread, IF it's not instantied as a pointer.

controller.add(&hisThread);

controller.add(&sensorReadings);

...

or

// Instantiate a new StaticThreadController with the number of threads to be supplied as template parameter

StaticThreadController<3> controller (&myThread, &hisThread, &sensorReadings);

// You don't need to do anything else, controller now contains all the threads.

...

* You have created, configured, grouped it. What is missing? Yes, whe should RUN it!

// call run on a Thread, a ThreadController or a StaticThreadController to run it

controller.run();

This will run all the Threads that NEED to be runned.

Congratulations, you have learned the basics of ArduinoThread. If you want some TIPS, see bellow.

**TIPs and Warnings**

* ThreadController is not a LinkedList. It's "MAXIMUM" size (the maximum Threads that it can store) is defined on ThreadController.h (default is 15)
* !!!! VERY IMPORTANT !!!! When extending Thread class and implementing the function run(), always remember to put runned(); after all, otherwhise the Thread will ALWAYS run.
* It's a good idea, to create a Timer interrupt and call a ThreadController.run() there. That way, you don't need to worry about reading sensors and doing time-sensitive stuff on your main code (loop). Check ControllerWithTimer example.
* Inheriting from Thread or even ThreadController is always a good idea. For example, I always create base classes of sensors that extends Thread, so that I can "register" the sensors inside a ThreadController, and forget about really reading sensors, just getting theirs values within my main code. Checkout SensorThread example.
* Remember that ThreadController is in fact, a Thread. If you want to enable or disable a GROUP of Threads, think about putting all of them inside a ThreadController, and adding this ThreadController to another ThreadController (YES! One ThreadController inside another). Check ControllerInController example.
* There is a StaticThreadController which is better to use when you know exact number of threads to run. You cannot add or remove threads in runtime, but StaticThreadController doesn't have additional memory overhead to keep all the treads together, doesn't have any limitations how many threads to store (except of available memory) and also the code may be slighly more optimized because all the threads always exist and no need to do any runtime checks.
* Check the full example CustomTimedThread for a cool application of Threads that runs for a period, after a button is pressed.
* Running tasks on the Timer interrupts must be tought REALLY carefully

You cannot use "sleep()" inside a interrupt, because it will get into a infinite loop.

Things must do stuff quickly. Waiting too loooong on a interrupt, means waiting too loooong on the main code (loop)

Things might get "scrambled". Since Timers interrupts actualy "BREAK" your code in half and start running the interrupt, you might want to call noInterrupts and interrupts on places where cannot be interrupted:

noInterrupts();

// Put the code that CANNOT be interrupted...

interrupts(); // This will enable the interrupts egain. DO NOT FORGET!

**Library Reference**

**You should know:**

* bool Thread::enabled - Enables or disables the Thread. (do not stop it from running, but will return false when shouldRun() is called)
* void Thread::setInterval() - Setts the desired interval for the Thread (in Ms).
* bool Thread::shouldRun() - Returns true, if the Thread should be runned. (Basicaly,the logic is: (reached time AND is enabled?).
* void Thread::onRun(<function>) - The target callback function to be called.
* void Thread::run() - This will run the Thread (call the callback function).
* int Thread::ThreadID - Theoretically, it's the address of memory. It's unique, and can be used to compare if two threads are identical.
* int Thread::ThreadName - A human-redable thread name. Default is "Thread ThreadID" eg.: "Thread 141515"; Note that to enable this attribute, you must uncomment the line that disables it on 'Thread.h';
* protected: void Thread::runned() - Used to reset internal timer of the Thread. This is automaticaly called AFTER a call to run().
* void ThreadController::run() - This will run the all Threads within the ThreadController, only if needed (if shouldRun returns true);
* bool ThreadController::add(Thread\* \_thread) - This will add a the thread to the ThreadController, and return true if suceeded (it the array is full, returns false).
* void ThreadController::remove(Thread\* \_thread) - This will remove the Thread from the ThreadController.
* void ThreadController::remove(int index) - This will remove the thread on the position index.
* void ThreadController::clear() - This will remove ALL threads from the ThreadController array.
* int ThreadController::size(bool cached = true) - Returns how many Threads are allocated inside the ThreadController. If cached is false, will force the calculation of threads.
* Thread\* ThreadController::get(int index) - Returns the Thread on the position index.
* void StaticThreadController::run() - This will run the all Threads within the StaicThreadController, only if needed (if shouldRun returns true);
* int StaticThreadController::size() - Returns how many Threads are allocated inside the StaticThreadController.
* Thread\* ThreadController::get(int index) - Returns the Thread on the position index and nullptr if index is out of bounds.