TaskManager Quickly

A Fast Introduction to TaskManager

Version 1.0

# Overview

TaskManager is a simple task swapper for Arduino. It allows a developer to replace the simple loop(){} structure with multiple loop-like tasks. TaskManager contains its own loop(){}, which proceeds to call the individual user-loop tasks on a round-robin basis.

This book is meant to quickly introduce TaskManager. It covers all of the core features, including base functional notes. It does not have a lot of examples – see the tutorial book for in-depth examples.

A simple example, with two blinking lights, is

//

// Blink two LEDs at different rates

// Uses TaskManager

// Uses YieldDelay to reschedule with delays

//

#include <TaskManager.h>

#define LED\_1 3

#define LED\_2 4

void setup() {

// set up the LEDs

pinMode(LED\_1, OUTPUT);

digitalWrite(LED\_1, LOW);

pinMode(LED\_2, OUTPUT);

digitalWrite(LED\_2, LOW);

TaskMgr.add(1, loop\_led\_1);

TaskMgr.add(2, loop\_led\_2);

}

void loop\_led\_1() {

static bool led\_1\_state = LOW;

led\_1\_state = (led\_1\_state==LOW) ? HIGH : LOW;

digitalWrite(LED\_1, led\_1\_state);

TaskMgr.yieldDelay(700);

}

void loop\_led\_2() {

static bool led\_2\_state = LOW;

led\_2\_state = (led\_2\_state==LOW) ? HIGH : LOW;

digitalWrite(LED\_2, led\_2\_state);

TaskMgr.yieldDelay(400);

}

Some basic notes on this program:

1. There is a setup() procedure, just like any other Arduino program.
2. There is no loop() procedure. TaskManager provides its own loop(). In normal Arduino programs, the user has only a single task, called loop().
3. TaskManager provides a single object, TaskMgr, to manage all of the tasks. You will use various TaskMgr methods to perform task-related operations.
4. There are two procedures that represent independent tasks: loop\_led\_1() and loop\_led\_2(). These both look like the normal “blink” program, just for different LEDs.
5. You use   
   #include <TaskManager.h>  
   to use TaskManager.
6. To add a procedure to TaskManager’s task list, use  
   TaskMgr.add(tasked, procedure);  
   Each task requires a unique ID in the range [1 239].
7. Instead of calling delay(ms), call TaskMgr.yieldDelay(ms). The delay() routine halts the entire program. The TaskMgr.yieldDelay() will suspend the single task for the specified period, allowing other tasks to run.

Some of the key attributes of TaskManager are

1. The number of tasks is limited by program memory. Each task consumes around 54 bytes of RAM while running.
2. Tasks are independent, and swap on a round-robin basis. Tasks will execute repeatedly unless they are waiting for something.
3. Tasks can communicate with messages (sending data) or signals (a simple “nudge”, no data sent between tasks).
4. Tasks can run on different Arduinos connected with RF24 transceivers. Tasks can send messages and signals between nodes using the RF24 transceivers.

To use TaskManager,

* #include <TaskManager.h>  
  in your main program.
* Use setup(){} to configure the hardware.
* Do not write loop() {}
* If you have secondary .cpp files (your source code is in several files) and you are using TaskManager routines in these files,   
  #include <TaskManagerSub.h>  
  in the .cpp files instead of TaskManager.h
* Write each task in its own void myTask(){} procedure. Think of each myTask as its own independent loop().
* Add each task using a TaskMgr.add\*() routine in setup().

# Simple Tasks

A task is a user activity. It is a single procedure with no parameters. You use one of the add\*() routines to add tasks to TaskManager’s run-list.

After a task exits, it is automatically re-scheduled. Optionally, a task may use one of the yield\*() routines to exit (and reschedule) from any point within the routine.

When a task is restarted, it restarts at the TOP of the routine (even if it yield\*()’ed from the middle). However, the TM macros can be used to get around this by forcing a restart from the last yield-point. The TM macros are described in a different section.

At present, there is no method of suspending or killing tasks. These may be added in a future version.

There is no method of prioritizing tasks.

## Adding Tasks

A task is added with one of the TaskMgr.add\*() routines. The following are required parameters:

1. A unique task ID. This is a single byte in the range 0 through 239. Tasks 0 and 240 through 255 are reserved for internal use.
2. The name of the task to be executed.

The add\*() routines are listed below. See the section on signals and messages for more information on signaling and messaging.

The addWait\*() routines will add tasks that start in a suspended state. They will activate upon the given conditions. After processing, they will reschedule themselves to run immediately unless a special yield\*() method is used to exit.

The addAutoWait\*() routines will add tasks that start in a suspended state, and will activate upon the given conditions. After processing, they will reschedule themselves using the same suspension method unless a special yield\*() method is used to exit.

* TaskMgr.add(byte taskID, void (\*task)())
  + add a task that will automatically reschedule itself to run immediately (as soon as possible) after completion.
* TaskMgr.addWaitDelay(byte taskID, void (\*task)(), unsigned long msDelay)  
  TaskMgr.addAutoWaitDelay(byte taskID, void (\*task)(), unsigned long msDelay)
  + Add a task that will delay for msDelay before starting.
* TaskMgr.addWaitUntil(byte taskID, void (\*task)(), unsigned long msTime)
  + Add a task that will wait until system clock time msTime before starting. Not really a useful routine.
* TaskMgr.addWaitMessage(byte taskID, void (\*task)(), unsigned long timeout=0)  
  TaskMgr.addAutoWaitMessage(byte taskID, void (\*task)(), unsigned long timeout=0)
  + Add a task that is suspended, waiting for a message. It will wait up to timeout ms (0==forever).
* TaskMgr.addWaitSignal(byte taskID, void(\*task)(), byte sigNum, unsigned long timeout=0)  
  TaskMgr.addAutoWaitSignal(byte taskID, void(\*task)(), byte sigNum, unsigned long timeout=0)
  + Add a task that is suspended, waiting for a specified signal. It will wait up to timeout ms (0==forever).

## Yielding

Yielding exits a task, allowing other tasks to execute. TaskMgr.yield\*() can be called from any point within a routine.

Using yield() will reschedule the task using its default approach. If TaskMgr.add() had been used to create the task, for example, it will run again as soon as possible. If TaskMgr.addAutoWaitSignal() had been used, the task will wait for a signal.

Note that yieldDelay(0) can be used to force an addAutoWaitSignal() task to resume immediately (instead of waiting for a signal).

As a reminder, the task will resume from the start of the routine. If you desire to resume from the next statement after the yield(), use one of the TM macros.

If a procedure returns with a return statement or by falling out through the bottom, it will act as if it had ended with TaskMgr.yield().

There is a very subtle difference between using TaskMgr.AddAutoWaitDelay(ms) as compared to using TaskMgr.Add() and TaskMgr.yieldDelay(ms). In the first case, it will attempt to restart the procedure ms milliseconds after the last START time. That is, it will try to run the task every ms milliseconds. In the second case, it will try to restart the procedure ms milliseconds from the time the procedure YIELDED.

The following are the different yield\*() routines:

* TaskMgr.yield() – return to the TaskManager scheduler, and reschedule the task using its default scheduling method.
* TaskMgr.yieldDelay(unsigned long int msDelay) – return to the TaskManager scheduler, and don’t reschedule the task for msDelay ms.
* TaskMgr.yieldUntil(unsigned long int msWhen) – return to the TaskManager scheduler, and don’t reschedule until the system clock is at or after the listed time.
* TaskMgr.yieldForSignal(byte sigNum, unsigned long int timeout=0) – return to the TaskManager scheduler, and don’t reschedule until a given signal has arrived. If timeout is not specified, it will wait for the signal. If timeout is specified (and is greater than zero), the task will be scheduled if ‘timeout’ ms have passed without a signal arriving.
* TaskMgr.yieldForMessage(unsigned long int timeout=0) – return to the TaskManager scheduler, and don’t reschedule until a message has arrived. If timeout is not specified, it will wait for the mesage. If timeout is specified (and is greater than zero), the task will be scheduled if ‘timeout’ ms have passed without a message arriving.

# Simple Messages and Signals

Messages and signals allow tasks to communicate in a structured manner. Here are the key points of messages and signals:

* A message is a block of data transmitted between tasks. It can have up to 29 bytes of information. This can be binary data or a (null terminated) string. It is defined by its content.
* A signal is a content-free event. It is defined by its signal number, an 8-bit unsigned value in the range [1 239]. Signals 0 and [240 255] are reserved for internal use.
* A message is sent from one task to another task.
  + To receive a message, a task yields for a message. This can either be done explicitly (yieldForMessage()) or by using addAutoWaitMessage().
  + To send a message, you use sendMessage(taskID, message).
* A signal is sent from one task to another task.
  + To receive a signal, a task yields for a signal. This can either be done explicitly (yieldForSignal()) or by using addAutoWaitSignal().
  + To send a signal, you use sendSignal(sigNum) or sendSignalAll(sigNum). The former sends to the next schedulable task after the current one waiting for signal sigNum. The latter sends to all tasks waiting for signal sigNum.
* When a task sets itself to wait for a message or signal, it can also set a timeout (in ms). If the timeout arrives before the message or signal does, the task is automatically activated.
* A task can find out who sent it a message or signal by using getSource().
* A task can find out whether or not it timed out by using timedOut().
* A task can retrieve its message by using getMessage().
* If there is no task waiting for a signal or a message is sent to a nonexistent task, the signal/message is discarded.
* If multiple messages or signals are sent to a task before the task starts, only the last message or signal will be saved. The others are lost. There is no “lost message” indicator.

The methods of waiting for messages and signals have been described in the sections *Adding Tasks* and *Yielding*. The methods of sending messages and signals are:

* TaskMgr.sendSignal(byte sigNum) – send a simple signal to the first task that is waiting for it.
* TaskMgr.sendSignalAll(byte sigNum) – send a simple signal to every task that is waiting for it.
* TaskMgr.sendMessage(byte taskID, void\* buf, int len) – send an arbitrary message of up to 29 bytes to the stated task. This is used to pass structures.
* TaskMgr.sendMessage(byte taskID, char\* message) – send a string message of up to 29 bytes (including the null terminator) to the stated task.

Additional methods are:

* TaskMgr.getSource(byte& fromTaskID) – return the taskID of the node that sent a signal or message. Used to send replies to the correct originator.
* TaskMgr.timedOut() – return true if the routine was invoked because it timed out before receiving a signal/message. Returns false if there was a legitimate signal/message
* TaskMgr.getMessage() – returns a void\* pointer to the message. Note this is an actual pointer to the internal message buffer for the task. The program is responsible for copying out the data. The buffer may be overwritten upon receipt of a new message.

# TaskManager and RF

TaskMgr has the ability to run independently on separate Arduino nodes, connected by RF24 modules. This allows tasks on separate nodes to communicate through signals and messages.

It is basically the same TaskManager as before. The only core difference is that sendSignal(), sendSignalAll(), and sendMessage() can now send to tasks on different nodes.

Wiring up an RF24 transceiver is done as follows:

* The RF24 has eight lines:
  + Power (V+ and GND)
  + MISO and MOSI
  + ….

To use TaskManagerRF,

* Use  
  #include <TaskManagerRF.h>  
  instead of   
  #include <TaskManager.h>
* If you use TaskManagerSub.h, use TaskManagerRFSub.h instead.
* Before the  
  #include <TaskManagerRF.h>  
  (or TaskManagerRFSub.h), include the following:  
  #include <SPI.h>  
  #include <RF24.h>
* Use the TaskMgr variable normally. It will now include RF code.

TaskManagerRF adds the following routines:

* TaskMgr.radioBegin(byte nodeID, byte cePin, byte csPin) – start the radio receiver, declaring this to be node nodeID. The RF24 transceiver will need to be wired to the stated ce and cs pins.
* TaskMgr.sendSignal(byte nodeID, byte sigNum) – like sendSignal(sigNum) except it will send the signal to a different node.
* TaskMgr.sendSignalAll(byte nodeID, byte sigNum) – like sendSignalAll(sigNum) except it will send the signal to a different node.
* TaskMgr.sendMessage(byte nodeID, byte taskID, void\* buf, int len) – like sendMessage(taskID, buf, len) except it will send the message to a different node.
* TaskMgr.sendMessage(byte nodeID, byte taskID, char\* message) – like sendMessage(taskID, message) except it will send the message to a different node.
* TaskMgr.getSource(byte& fromNodeId, byte7 fromTaskID) – like getSource(fromTaskID) except it will also return the node of the source. If the source is the same as the current node, fromNodeId will be set to 0. Note that getSource(fromTaskID) will return just the task even if the signal/message came from a different node.
* TaskMgr.myNodeId() – returns the ID of this node.

# TaskManager Macros

The one downside to the various TaskMgr.yield\*() routines is that execution will always restart at the top of the task code. It will not resume at the next statement.

To get around this, the TaskManager macros were written. They allow a task to continue at the next statement after executing a TaskMgr.yield\*() call.

The TaskManager macros are ugly, and the code they produce is even uglier. The following macros are defined for use in TaskManager tasks:

* TM\_INIT() – start the macro processing.
* TM\_YIELD(id) – perform a TaskMgr.yield() operation.
* TM\_YIELDDELAY(id, msDelay) – perform a TaskMgr.yieldDelay() operation.
* TM\_YIELDSIGNAL(id, sigNum) – perform a TaskMgr.yieldSignal(sigNum) operation.
* TM\_YIELDSIGNALTO(id, sigNum, timeout) – perform a TaskMgr.yieldSignal(sigNum, timeout) operation.
* TM\_CLEANUP() – end the macro processing

Very specific conditions need to be followed when using them:

* Use theses routines ONLY within the actual task procedure. Do not use them within subprocedures that are called by the task procedure.
* All variables (whose values you need to preserve across the yield()) must be static or external to the routine.
* Place TM\_INIT() at the start of the routine.
* Place TM\_CLEANUP() at the end of the routine.
* Use TM\_YIELD\*() wherever you need between the TM\_BEGIN() and TM\_CLEANUP().
* Do not use any of the TaskMgr.Yield\*() methods.
* The id must be a unique integer within the routine. I recommend using 1, 2, 3, …
* Note that the TM\_YIELD\*() calls can be placed inside of if and else blocks, loops, etc.
* The TM macros create a local variable, \_\_tmNext\_\_. Please do not use this variable.
* Code that is placed before TM\_INIT() will be executed after each TM\_YIELD\*(). Code that is placed after will TM\_INIT() only be executed (a) the first time the task runs, and (b) the next time the task runs after it exits at the bottom (after TM\_CLEANUP()).

Here is blinky, simplified and extended to do more flashing (showing off the macros):

void blinkyWithMacros() {

// short and three long blinks

static int i;

TM\_INIT();

digitalWrite(13, LOW);

TM\_YIELDDELAY(1, 250);

digitalWrite(13, HIGH);

TM\_YIELDDELAY(2, 250);

For(i=0; i<3; i++) {

digitalWrite(13, LOW);

TM\_YIELDDELAY(3, 1000);

digitalWrite(13, HIGH);

TM\_YIELDDELAY(4, 1000);

}

TM\_CLEANUP();

}

# Other Routines

# Other Stuff

## Changing A Task’s Yield Process

These routines allow you to change how a task will function after yielding. They are primarily used to set the yieldAuto\* processing.

* TaskMgr.setYield()  
  TaskMgr.setYield(byte taskID)
  + Set the task so it will reschedule to run as soon as possible after a normal yield().
* TaskMgr.setAutoWaitDelay(unsigned long msDelay)  
  TaskMgr.setAutoWaitDelay(byte taskID, unsigned long msDelay)
  + Set the task so it will reschedule to run after the specified delay, after a normal yield(). If the task is not running, it will now be waiting for msDelay ms from the current clock.
* TaskMgr.setAutoWaitMessage(unsigned long timeout=0)  
  TaskMgr.setAutoWaitMessage(byte taskID, unsigned long timeout=0)
  + Set the task so it will reschedule to wait for a message, after a normal yield(). If the task is not currently running, it will now be waiting for a message.
* TaskMgr.setAutoWaitSignal(byte sigNum, unsigned long timeout=0)  
  TaskMgr.setAutoWaitSignal(byte taskID, byte sigNum, unsigned long timeout=0)
  + Set the task so it will reschedule to wait for a signal, after a normal yield(). If the task is not currently running it will now be delayed until the signal has been received. If the task had been waiting for a different signal, it will be set to wait for the new signal instead of the prior one.

# Constants of Interest

## Miscellaneous Constants

* Maximum message length: 29 bytes

## Reserved Task IDs

* 0: Used to indicate “this task” under certain conditions
* 253: Ping Responder
* 254: Radio receiver
* 255: Null task

## Reserved Signal IDs

* 254: PingResponse
* 255: PingRequest

# Advanced Stuff

## Using the Same Procedure for Several Tasks

# Examples