TaskManager -- A Quick Summary

Sunday, July 26, 2015

1:13 PM

TaskManager is a lightweight, powerful, easy-to-use system that allows an Arduino to manage multiple concurrent interacting cooperative tasks. It offers the following functionality:

* Easy to use. It follows the Arduino "loop()" paradigm. If you can write one loop() to handle a task, you can write three procedures loop\_1(), loop\_2(), and loop\_3(), to handle your three tasks. Just as loop() was called over and over, your three procedures will be called over and over, in sequence. That's pretty much it to using the TaskManager library.
* Tasks can interact with each other. They can send signals and messages. They can wait for signals and messages.
* Arduinos can network, and tasks can communicate with tasks running on different systems (nodes). Just as a task can send a message to another task on the same Arduino node, it can send the same message to a task running on a different Arduino node. This allows large numbers (253!) of Arduinos to interact and interoperate.

Introduction

Saturday, July 18, 2015

10:45 AM

I like the concept of complex, interacting systems. Especially ones constructed of small nodes, where each node has limited functionality, and a collection of nodes can interact in complex, unusual, and surprising manners.

I also like programming small systems. I spent years using 8080/Z-80 based systems, where you had complete control of the environment. You could add and address hardware exactly as needed, and didn’t need to worry about massive overheads and potential interference by large operating systems.

CP/M, the entire operating system, weighed in at around 4K of code. Including low-level drivers, abstract device interfaces, full disk/file management system, and command line processor. Unix and Xenix (and since the 90s, Linux) are a lot larger. You could understand CP/M. The whole thing. Not \*nix.

As the design concepts for some of my projects got more complex, the simple setup/loop system became difficult to manage. And as I wanted activities on one node to communicate with activities on different nodes, it became unmanageable.

So I started to look at different task management systems for the Arduino. I found a number of published systems, but none met my needs. So I wrote one.

I outlined the core set of functionality and API approach based on a few key conditions:

1. It should be simple in design. It would not be an interrupt-driven preemptive operating system. It would be a simple cooperative system.
2. It should be simple to use. Simple tasks should be simple. It should be easy to learn and use.
3. It should support methods of signaling and message passing between tasks.
4. It should support (in subsequent versions) similar methods of signaling and message passing between tasks running on different nodes in a network.

Existing multitasking systems did not meet this need. Many were more complex, using classing and subclassing to defined tasks. None supported multi-node environments. Few had any form of communication between tasks.

So I wrote this one.

Acknowledgements

Saturday, July 18, 2015

5:49 PM

I'd like to thank Kyle Bradshaw and Michael Caldwell for being the first true tester. He provided valuable feedback, as well as providing some functioning sample code (and a functioning system) to get past a hurdle I had with the RF24 radio.

All circuit diagrams were made with fritzig (<http://fritzing.org/home/>).

Coding Style

Saturday, July 18, 2015

6:28 PM

Code isn't perfect, it is meant to be a clear example. There are better ways, but they aren't as clear. And the problems are still there.

Fonts in use:

* Text
  + This is sample text.
* Code
  + myVal = myVal \* 2 + 6;

TaskManager refers to the library, and is a C++ class. TaskMgr is an object of type TaskManager, and is the single object that you use to control all of your tasks.

Capability Overview

Saturday, July 18, 2015

5:37 PM

The TaskManager library allows the Arduino programmer to define and run a set of independent tasks on one (or a network of) Arduino processors. It offers the following

* The overall structure is very familiar to the standard setup()/loop() model.
* You can have up to 191 tasks running on a single processor (if they will fit).
* The tasks are run one after the other (round-robin). Each gets a turn. Each runs for as long as it wants, then gives back control so the next task can run.
* A task is normally enabled and will run when it has a chance. However, a task may also request not to be run until any of a variety of events have passed:
  + Until a certain amount of time has passed (similar to "delay()"),
  + Until it receives a signal from another task that it is allowed to run,
  + Until it receives a message (with some information) from another task, indicating that it should process the message
* A network may be constructed of more than one Arduino processors. Tasks may send signals or messages to any other task on any other node in the network. You can have up to 253 nodes.

Why TaskManager

Monday, July 13, 2015

4:57 PM

Technically, a library like TaskManager isn't necessary. Anything you could want to do, you can program through one complex loop() process along with an accompanying set of support procedures. However, in more complex cases, the interactions between time, data, and the processes, can make coordination difficult. TaskManager simplifies this.

This section presents variants of the basic "blinky" application. It will add functionality to "blinky", and show how TaskManager can simplify the program design in complex systems.

Keep in mind that "blinky" is a representative process. It represents a programmed activity that needs to be performed on a periodic basis. In the more complex version, it also represents a single Arduino node that is executing a larger number of time- and event-driven activities.

**Blinky**

This is the circuit we will be using for blinky. We are using a discrete LED with a resistor, all attached to D2.

Machine generated alternative text:


Here is the associated program code to have it blink at a 1Hz rate.

//

// Standard "blink" program

#define LED\_1 2

bool led\_1\_state;

void setup() {

pinMode(LED\_1, OUTPUT);

led\_1\_state = LOW;

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

}

void loop() {

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

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

delay(500);

}

It doesn't need much explanation. Note that the default "blinky" program (the one that is provided in the Arduino IDE example directory) has two delays and explicit digitalWrite(HIGH/LOW) statements. We replaced it with one delay to make later code simpler (only one delay in the loop() code).

Here is the TaskManager version.

//

// Standard "blink" program

#define LED\_1 2

bool led\_1\_state;

void setup() {

pinMode(LED\_1, OUTPUT);

led\_1\_state = LOW;

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

TaskMgr.add(1, myLoop);

}

void myLoop() {

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

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

TaskMgr.yieldDelay(500);

}

You don't need TaskManager for simple things. But here are the differences.

1. You need to #include a few files. You *will* need to #include the SPI.h and RF24.h. TaskManager includes networking with RF24 transcievers, even if you don't use this feature.
2. You don't have a loop() procedure.
3. You have a procedure for each task you are running. In this case, there is only one task, myLoop().
4. In setup(), you tell TaskManager what all of your tasks are. This is the statement TaskMgr.add(1, myLoop).
5. You don't use delay(). Use TaskMgr.yieldDelay() instead. There are other things to worry about with delays, but we will talk about them later.

**Blinky2**

This is the circuit we will be using for the rest of this section. It has three LED (on D2, D3, and D4) and one switch (on D5).

Machine generated alternative text:
fritzlng 
Ill-Ill 
0 

We will write blinky2 for the case where we wanted one process that executed twice per second (once to flash high, once to flash low), and another process that executed ten times per second (high low high low…). This code does not use the switch or third LED. We'll use them later.

This is the code we want to write:

#define LED\_1 2

#define LED\_2 3

bool led\_1\_state;

bool led\_2\_state;

void setup() {

pinMode(LED\_1, OUTPUT);

led\_1\_state = LOW;

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

pinMode(LED\_2, OUTPUT);

led\_2\_state = LOW;

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

}

void loop() {

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

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

delay(500);

}

void loop2() {

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

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

delay(100);

}

Unfortunately, we can't. We can only have one loop(). And we can't stick both of the sets of code shown into a single loop() because the timing will overlap -- we would delay 600ms between cycles of loop, and the flashing would be All Wrong.

Instead, we write code that keeps track of when each light last flashed. The following would do the trick:

//

// Standard "blink" program

#define LED\_1 2

long int last\_1\_time;

bool led\_1\_state;

#define LED\_2 3

long int last\_2\_time;

bool led\_2\_state;

void setup() {

pinMode(LED\_1, OUTPUT);

pinMode(LED\_2, OUTPUT);

last\_1\_time = millis();

last\_2\_time = last\_1\_time;

digitalWrite(LED\_1, LOW);

digitalWrite(LED\_2, LOW);

led\_1\_state = LOW;

led\_2\_state = LOW;

}

void loop() {

long int now = millis();

if(now-last\_1\_time > 500) {

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

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

last\_1\_time = millis();

}

if(now-last\_2\_time > 100) {

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

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

last\_2\_time = millis();

}

}

We won't go into the loop in-depth. Basically, for each light, it keeps track of when it last flashed. If enough time has passed, it changes the state and updates the "last flashed" time.

Now, it is time make the system even uglier. We are adding in that third LED and the switch. We will add the external event (the switch), and let it control the third LED. Here is the final, TaskManager-free code.

//

// Blink two LEDs at different rates

// Respond to a switch on the third LED

//

// first LED

#define LED\_1\_PORT 2

long int last\_1\_time;

bool led\_1\_state;

// second LED

#define LED\_2\_PORT 3

long int last\_2\_time;

bool led\_2\_state;

// third LED and its switch

#define LED\_3\_PORT 4

#define SWITCH\_PORT 5

bool led\_3\_state;

bool switchIsPressed;

void setup() {

// set up first LED

pinMode(LED\_1\_PORT, OUTPUT);

digitalWrite(LED\_1\_PORT, LOW);

led\_1\_state = LOW;

last\_1\_time = millis();

// set up second LED

pinMode(LED\_2\_PORT, OUTPUT);

digitalWrite(LED\_2\_PORT, LOW);

led\_2\_state = LOW;

// set up switch and third LED

pinMode(LED\_3\_PORT, OUTPUT);

pinMode(SWITCH\_PORT, INPUT\_PULLUP);

digitalWrite(LED\_3\_PORT, LOW);

led\_3\_state = LOW;

switchIsPressed = false;

// log the start times of the first two LEDs

last\_1\_time = millis();

last\_2\_time = last\_1\_time;

}

void loop() {

// get the current time

long int now = millis();

// see if enough time has passed to flip the first LED

if(now-last\_1\_time > 500) {

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

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

last\_1\_time = millis();

}

// see if enough time has passed to flip the second LED

if(now-last\_2\_time > 100) {

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

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

last\_2\_time = millis();

}

// if the switch has been pressed, invert the LED and debounce

// if it has been released, just debounce

if(digitalRead(SWITCH\_PORT)==LOW/\*==pressed/closed\*/) {

if(!switchIsPressed) {

// someone pressed the switch

led\_3\_state = (led\_3\_state==LOW) ? HIGH : LOW;

digitalWrite(LED\_3\_PORT, led\_3\_state);

switchIsPressed = true; // debounce

delay(50);

}

} else if(switchIsPressed /\* we know switchPort==HIGH/open \*/) {

switchIsPressed = false;

delay(50); // debounce

}

}

Ugh. The loop() code is ugly. Each individual component is complex. All we really wanted was three simple blocks of code, one per process.

**blinky2\_tm**

The following block shows the TaskManager version of the application

It has the following key points:

* There is a single object called TaskMgr. Note the difference between the object TaskMgr and its underlying class definition, TaskManager. Although we refer to the TaskManager library, all of our calls are made through the TaskMgr object. This is very similar to using the Serial object to print things to the serial port.
* There are a small number of additional #include statements.
* The port initialization code is the same.
* There are three routines, loop\_led\_1(), loop\_led\_2(), and loop\_led\_3(). Each has ONLY the code for a single activity. Each looks like an independent loop() procedure.
* *The program does not have a loop() procedure!*
* There are three lines of code in the setup() procedure to tell TaskMgr that it needs to manage the three tasks.
* We don't use delay(). Instead, we use TaskMgr.yieldDelay().

//

// Blink two LEDs at different rates

// Respond to a switch on the third LED

//

#include <SPI.h>

#include <RF24.h>

#include <TaskManager.h>

#define LED\_1\_PORT 2

bool led\_1\_state;

#define LED\_2\_PORT 3

bool led\_2\_state;

#define LED\_3\_PORT 4

#define SWITCH\_PORT 5

bool led\_3\_state;

bool switchIsPressed;

void setup() {

pinMode(LED\_1\_PORT, OUTPUT);

digitalWrite(LED\_1\_PORT, LOW);

led\_1\_state = LOW;

pinMode(LED\_2\_PORT, OUTPUT);

digitalWrite(LED\_2\_PORT, LOW);

led\_2\_state = LOW;

pinMode(LED\_3\_PORT, OUTPUT);

pinMode(SWITCH\_PORT, INPUT\_PULLUP);

digitalWrite(LED\_3\_PORT, LOW);

led\_3\_state = LOW;

switchIsPressed = false;

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

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

TaskMgr.add(3, loop\_led\_3);

}

void loop\_led\_1() {

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

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

TaskMgr.yieldDelay(500);

}

void loop\_led\_2() {

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

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

TaskMgr.yieldDelay(100);

}

void loop\_led\_3() {

if(digitalRead(SWITCH\_PORT)==LOW/\*==pressed/closed\*/) {

if(!switchIsPressed) {

// someone pressed the switch

led\_3\_state = (led\_3\_state==LOW) ? HIGH : LOW;

digitalWrite(LED\_3\_PORT, led\_3\_state);

switchIsPressed = true;

TaskMgr.yieldDelay(50); // debounce

}

} else if(switchIsPressed /\* we know switchPort==HIGH/open \*/) {

switchIsPressed = false;

TaskMgr.yieldDelay(50); // debounce

}

}

**How This Works**

In a nutshell, here are the key points of this program. The next two sections will present the core of TaskManager's operation, and the following three sections will present additional methods for task control and interaction.

* TaskMgr supplies loop(). You will note this program does not include a loop() procedure. Instead, it has been replaced by a group of processes (loop\_led\_1(), loop\_led\_2(), and loop\_led\_3()), each of which operates like an independent "loop()"-like program.
* TaskMgr's loop() does the task management work. TaskMgr's loop() alternates among all of the tasks it controls. It provides the mechanism for process control and process interaction.
* You inform TaskMgr of what tasks you want executed through TaskMgr.add(). TaskMgr.add(id, fn) tells TaskMgr that fn is one of the things it will control. Each task has an id (an integer in the range 1..192). Each should be unique, but this is not necessary.
* TaskMgr.yieldDelay() stops the routine and guarantees it will not be restarted until at least ms milliseconds have passed. Note there are some subtle differences between delay() and TaskMgr.yieldDelay(). They will be described in the next two sections.

**What's Next**

The remainder of this section describes the whole of the TaskManager library. The sections cover

* Basics. How TaskManager runs and manages tasks, how to add tasks, and how to perform simple delay()-like operations.
* Signaling and Messaging. How information and basic control can be passed between tasks.
* Auto-Rescheduling. More complex variants on the add() routine that make life simpler.
* Networking. How tasks running on different nodes in a network can communicate.
* Programmer's summary. The complete set of available operations in the TaskManager library, summarized in one place

Task Manager Basics

Monday, July 13, 2015

4:59 PM

The basic Arduino system creates an environment where there is a single task (loop()) that is repeatedly called. Outside of your Arduino program, there is a main program that supplies the following functionality:

void main() {

setup();

while(true) {

loop();

}

}

The TaskManager library creates an environment where you can define many processes, and have them executed in sequence. TaskManager supplies its own loop(); you need to inform it what tasks to execute.

This section covers the basics of using TaskManager. It describes

* How to tell it what tasks to run,
* How to write the tasks (style guide),
* How to perform simple delaying operations (which are fairly common).

As a reminder, although we speak of the TaskManager library, all operations are performed using methods of the TaskMgr object.

**Adding Tasks**

All tasks are added using variants of the TaskMgr.add() routine. You call TaskMgr.add() during setup once for each task being added. Adding is normally performed during setup().

void TaskMgr.add(byte taskID, void (\*(task))() );

The taskID should be a value in the range 1..192. It identifies particular tasks on the system, and is used when sending messages, pausing and resuming tasks, etc. It should be unique -- you should not have several tasks with the same taskID. However, uniqueness is not verified.

Example:

void myTask() {

If(digitalRead(10)==HIGH) {

digitalWrite(11, HIGH);

}

}

void setup() {

TaskMgr.add(3, myTask);

}

Note that you will usually either have to write your procedure before setup() or use a forward declaration. The following would have worked equally well.

void myTask();

void setup() {

TaskMgr.add(3, myTask);

}

void myTask() {

If(digitalRead(10)==HIGH) {

digitalWrite(11, HIGH);

} else if (digitalRead(10)==LOW) {

digitalWrite(11, LOW);

}

}

**Writing Tasks**

Each task should be designed to run as quickly as possible. One call to the task should perform a single activity and exit. It should run in milliseconds and then return.

The procedure myTask() just presented is a good example. It tests an input and performs an action. When the activities for a single pass are done, the task procedure may exit in one of a number ways:

1. Exit out of the bottom of the procedure. The procedure will run again normally at its next scheduled turn. (Special side-note: the addAuto\*() routines modify this, and will be discussed in the next section.)
2. Use a return; statement anywhere in the procedure. This acts the same as exiting out of the bottom.
3. Use a TaskMgr.yield(); statement anywhere in the procedure. This acts the same as exiting out of the bottom.
4. Use a TaskMgr.yield\*(); statement anywhere in the procedure. This will cause the procedure to exit, and it will not be run again until after the appropriate condition has been met. We will describe one such form now (TaskMgr.yieldDelay()) and will describe other forms in the next section.

An important note: Regardless of how a task exits, the next time it runs, it will start its execution at the top. It will not resume execution after a yield\*() operation.

**Using yieldDelay() -- Replacing delay()**

Do not use delay() in a task. It will stop everything. Not only will the current task be stopped, but all other tasks will not get to use the time. Instead, TaskMgr has a method called yieldDelay(int ms) that has a similar effect. The yieldDelay(ms) procedure will do the following:

1. It will return from the current task.
2. It will guarantee that the task will not be called again for (at least) ms milliseconds.

The current task will be blocked, but other tasks will be allowed to run and make use of the time.

Using yieldDelay() and other yield functions calls for a slightly different coding style. The original blinky demonstrates this nicely. Its code looked like this:

void loop() {

digitalWrite(13, HIGH); // turn the LED on (HIGH is the voltage level)

delay(500); // wait for a second

digitalWrite(13, LOW); // turn the LED off by making the voltage LOW

delay(500); // wait for a second

}

You can't replace the delay(500); calls with TaskMgr.yieldDelay(1000);, even though both cause delays of 500ms. The delay() causes a delay, then returns to the next statement. yieldDelay() causes a return, and the code resumes at the TOP of loop, not the next statement.

Instead, in the last section, we re-coded the loop to look like this:

bool led\_1\_state;

void loop() {

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

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

delay(500);

}

The new code uses a variable to store the state, and performs the following:

1. Update the state
2. Write the data (perform the action)
3. Delay
4. Return

This naturally turns into the actual code using TaskManager:

bool led\_1\_state;

void loop\_led\_1() {

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

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

TaskMgr.yieldDelay(500);

}

This keeps the key principle of using yield\*(): The yield\*() call will be the last thing executed, and if needed, the routine will keep track of what code needs to be executed next.

An alternative version (with the same result!) looks like this.

void loop\_led\_1() {

If(led\_1\_state==LOW) {

Led\_1\_state = HIGH;

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

TaskMgr.yieldDelay(500);

} else {

Led\_1\_state = LOW;

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

TaskMgr.yieldDelay(500);

}

}

You normally wouldn't write this exact code this way, but this general form accomplishes the following:

1. It selects from a number of possible paths
2. It performs an action
3. It selects what the next step will be
4. It then yields

The yield\*() operation might be different at each step. If you had several LEDs you might be changing different ones (think of a stop light). And you might be executing things out of order or switching the order based on other inputs (a switch, for example).

A Fun Example

Wednesday, July 22, 2015

3:56 PM

This is a circuit I built to (a) run a real multi-tasking program on an Arduino, and (b) have some fun with RGB LEDs.

In the circuit, there are two RGB LEDs. Each is cycling colors differently. The program uses pulse width modulation (PWM) to vary the intensities of the red, green, and blues lines.

One LED is controlled by three tasks. Each task controls one line (red, green, blue). Each line goes from 0 to 255 and back to 0 at different rates (3, 5, and 7 second cycle times).

The second LED is controlled by a single task. It uses the HSV color system (hue, saturation, value). It keeps the colors fully saturated (e.g., strong red instead of pinkish) and with high value (bright instead of fading to black). It varies the hue from red to yellow to green to cyan to blue to magenta and back to red. This is a HSV diagram from a color picker. The saturated colors are along the top edge, desaturating as they go down to the bottom. The separate column on the right show value ranges from white to black.

Machine generated alternative text:


Here is how a Nano is wired for this :

Machine generated alternative text:


Here is the code. It does not include the code that converts HSV values (each of H, S, V in the range 0..255) to RGB values (also in the range 0..255). The HSV code is included in the distributed example.

It looks more complex than it actually is. There are four scheduled tasks, red(), green(), blue(), and hsv(). The first three call updater() to update pins as needed. That's about all. And it is pretty to watch.

/\*

Manipulate an RGB LED on pins 9(R), 10(G), 11(B) using PWM.

Ramps each color up and down at different speeds.

Uses the TaskManager library to manage independent tasks for each pin.

Also manipulates an RGB LED on pins 3(R), 5 (G), 6(B) using PWM.

Ramps H:[0 255] S=255 V=255, converting to RGB. A hue circle, essentially

SM Platt, 2014-10-20

\*/

#include <SPI.h>

#include <RF24.h>

#include <TaskManager.h>

#include <rgb\_hsv.h>

#include "Streaming.h"

/\* 10/21/14 Added code for second RGB LED that just traverses the H range \*/

// LEDs and pins

#define LED1R 9

#define LED1G 10

#define LED1B 11

#define LED2R 3

#define LED2G 5

#define LED2B 6

// KEEP RGB global so the reporter can access the values

int rLevel=1;

int gLevel=1;

int bLevel=1;

int hLevel=0;

// Task to manipulate the HSV cone on pins 3, 5, 6

void hsv() {

// Note that S and V will always be 255 since we are going on the saturated outer

// rim of the HSV cone

byte r, g, b;

byte h, s, v;

hsv\_to\_rgb(hLevel, 255, 255, r, g, b);

analogWrite(LED2R, r);

analogWrite(LED2G, g);

analogWrite(LED2B, b);

hLevel = hLevel==255 ? 0 : hLevel+1;

// if(hLevel>166)

// Serial << "h: " << hLevel << " -> rgb: " << r << ' ' << g << ' ' << b << '\n';

}

// Task to report the current RGB levels

void reporter() {

Serial << "rgb is [" << rLevel << ' ' << gLevel << ' ' << bLevel

<< "] hsl is [" << hLevel << " 255 255]\n";

}

// Common update procedure for each of the R, G, B tasks

// Note that this procedure takes around 512 cycles to complete.

void updater(unsigned int pin, int& val) {

analogWrite(pin, val<256 ? val : 511-val);

val = (val + 1);

if(val>511) val=1;

}

// The R, G, B tasks.

void red() {

updater(LED1R, rLevel);

}

void green() {

updater(LED1G, gLevel);

}

void blue() {

updater(LED1B, bLevel);

}

// Setup.

// Initialize the serial line.

// Add the tasks at their set rates

void setup()

{

Serial.begin(9600);

Serial << "Hello World xxx\n";

pinMode(3,OUTPUT);

pinMode(5,OUTPUT);

pinMode(6,OUTPUT);

pinMode(9,OUTPUT);

pinMode(10,OUTPUT);

pinMode(11,OUTPUT);

// Red runs every 3 seconds. Green every 5 seconds. Blue every 7 seconds

// There are 52 steps per the update procedure.

TaskMgr.addAutoWaitDelay(1, red, (3\*1000)/512);

TaskMgr.addAutoWaitDelay(2, green, (5\*1000)/512);

TaskMgr.addAutoWaitDelay(3, blue, (7\*1000)/512);

TaskMgr.addAutoWaitDelay(4, hsv, (20\*1000)/256); // 5 second loop

//TaskMgr.addAutoWaitDelay(5, reporter, 1000);

}

**Side notes**

I have found that the R, G, and B on the RGB LEDs I use are not balanced. That is, the three aren't all the same intensity, so if you write (255 255 255), you do not get pure white. However, by adjusting the resistors on each line, I can get a more balanced white. Here are the resistors I use:

|  |  |
| --- | --- |
| R | 100 |
| G | 288 |
| B | 913 |

The RGB LEDs I use are not frosted, and you can clearly see the individual R, G, and B components. This doesn't always make for good color mixing. To get around this, I use 3/4" acetal spheres (Small Parts/Amazon). I drill 6mm holes partway in. This is just large enough to hold the 5mm LED and its slightly wider base. I recommend using a drill press for this.

Signals and Messages

Monday, July 13, 2015

5:10 PM

TaskManager offers two methods for tasks to communicate with each other: Signals and Messages.

**Signals**

A *signal* is a simple global semaphore. Each signal is identified by a value in the range 0..255. As with taskIDs, user tasks should use signals in the range 1..191. Signal 0 and signals in the range 192..255 are reserved for internal (TaskManager) use.

A task that wants to suspend itself until some arbitrary event occurs will TaskMgr.yieldForSignal(byte signalID);. This will suspend the task until some other task executes TaskMgr.signal(byte signalID);.

Note that the signalID is *not* the taskID. They are two distinct values.

Note also that more than one task may be waiting for the same signalID. If the signaling task uses signal(), only one of the tasks will be signaled. If the signaling task uses signalAll(), all of the waiting tasks will receive a signal.

The key TaskManager signal methods are:

void yieldForSignal(byte signalID);

Return control to TaskMgr. The task will not be rescheduled until a different task has sent the listed signal.

void yieldForSignal(byte signalID, unsigned long int timeoutMs);

Return control to TaskMgr. The task will not be rescheduled until either a different task has sent the listed signal *or* the specified amount of time has passed. This allows a task to define an upper limit to how long it will wait for a signal.

void sendSignal(byte signalID);

Send a signal to the first task waiting for the listed signalID. The receiving tasks will be scheduled for execution and will run at the next available opportunity. The sending task will continue on the statement following the signal(). (This is not a yield\*().)

void sendSignalAll(byte signalID);

Send a signal to all tasks waiting for the listed signalID. The receiving tasks will be scheduled for execution and will run at the next available opportunity. The sending task will continue on the statement immediately following the sendSignal(). (This is not a yield\*().)

bool timedOut();

This tells whether or not a task was started because it has timed out. This is useful when the task had executed a yieldForSignal(signalID, timeoutMS) as it lets it know whether or not it had received the signal. The timedOut() function returns false if the task had received a signal and true if the time specified in yieldForSignal(signalID, timeoutMS) had passed before the signal had been received.

There is also a method for adding tasks in an already-waiting state.

void addWaitSignal(byte taskID, void (\*task)(), byte sigId, unsigned int timeoutMs=0);

Add the given task with the given taskID. Instead of being ready-to-run, it will be suspended as if it had already executed yieldForSignal(sigID, timeoutMs);. Note that timeoutMs is optional; if you do not specify a value (ex: addWaitSignal(taskID, task, sigID);) then it will be added without a timeout value.

**Messages**

A *message* is a packet of information that is passed between tasks. It can contain up to 30 bytes of information (or a string up to 29 characters long). A task can wait for a message, and a message is sent to a specific task. A message can only be sent to one task.

The key TaskManager messaging routines are:

void yieldForMessage();

Return control to TaskMgr. The task will not be rescheduled until a different task has sent it a message.

void yieldForMessage(unsigned long int timeoutMs);

Return control to TaskMgr. The task will not be rescheduled until either a different task has sent a message *or* the specified amount of time has passed. This allows a task to define an upper limit to how long it will wait for a message.

void sendMessage(byte taskID, char\* msg);

Send a message to the named task. The message is a null-terminated character string. The receiving task will be scheduled for execution and will run at the next available opportunity. The sending task will continue on the statement following the sendMessage(). (This is not a yield\*().) The receiving task can get the message with TaskMgr.getMessage().

void sendMessage(byte taskID, void\* data, int len);

Send a message to the named task. The message is a binary objec. The receiving task will be scheduled for execution and will run at the next available opportunity. The sending task will continue on the statement following the sendMessage(). (This is not a yield\*().) The receiving task can get the message with TaskMgr.getMessage().

void\* getMessage();

This routine is used by the procedure receiving the message. It returns a pointer to the message. It is up to the receiver to determine the content and structure of the message (binary data, character string).

bool timedOut();

This is the same timedOut() as described in Signals.

There is also a method for adding tasks in an already-waiting state.

void addWaitMessage(byte taskID, void (\*task)(), unsigned int timeoutMs=0);

Add the given task with the given taskID. Instead of being ready-to-run, it will be suspended as if it had already executed yieldForMessage(timeoutMs);. Note that timeoutMs is optional; if you do not specify a value (ex: addWaitMessage(taskID, task);) then it will be added without a timeout value.

**Example**

This example demonstrates the use of both signals and messages. It contains tasks that perform the following:

* button\_1: Watch a button, send a signal to two tasks to invert LEDs.
* led\_1a: Wait for a signal, then invert an LED.
* led\_1b: Wait for a signal, then invert an LED. If it does not receive a signal in 2 seconds, invert anyway.
* button\_2: Watch a button, send one of a set of messages to the message printer.
* messenger: Receive a message and print it to the serial line
* ticker: Once every five seconds, print a message with the time to the serial line

Here is the circuit for the program:

Machine generated alternative text:
◆ 
リ 廓 

Here is the program:

#include <Streaming.h>

#include <SPI.h>

#include <RF24.h>

#include <TaskManager.h>

/\*

Demonstrates different forms of signalling/messaging/yielding

Six tasks:

1. button\_1: watch a button, send signals to led\_1a and led\_1b

2. led\_1a: wait for signal, invert led when receives one

3. led\_1b: wait up to 2 seconds for a signal. Invert led when

receive signal or timeout

4. button\_2: send a message to messenger

5. messenger: receive a message and write it to the console

6. ticker: every 5 seconds write a message to the console

telling how many timeouts on led\_1b

\*/

// forward declarations

void button\_1();

void led\_1a();

void led\_1b();

void button\_2();

void messenger();

void ticker();

// our signals and delays

#define LED\_1A\_SIG 10

#define LED\_1B\_SIG 11

#define LED\_1B\_TIMEOUT 2000

#define TICKER\_DELAY 5000

// the ports

#define LED\_1A\_PORT 2

#define LED\_1B\_PORT 3

#define SWITCH\_1\_PORT 4

#define SWITCH\_2\_PORT 5

// some messages for messenger

int nextMessage = 0;

char\* theMessages[] = {

"Hello from TaskMgr!",

"This is the 2nd message.",

"Flash blit bing <done>",

"And now we restart",

"Next msg not sent - too long",

"This message is not sent -- it is too long",

"00000000001111111111222222222",

"01234567890123456789012345678",

"" // null message marks the end of the set

};

// global variables

int timeoutCounter; // how many led\_2 timeouts

void setup()

{

Serial.begin(9600);

// set up ports

pinMode(LED\_1A\_PORT, OUTPUT);

digitalWrite(LED\_1A\_PORT, LOW);

pinMode(LED\_1B\_PORT, OUTPUT);

digitalWrite(LED\_1B\_PORT, LOW);

pinMode(SWITCH\_1\_PORT, INPUT\_PULLUP);

pinMode(SWITCH\_2\_PORT, INPUT\_PULLUP);

// initialize globals

timeoutCounter = 0;

// add tasks

TaskMgr.add(1, button\_1);

TaskMgr.addWaitSignal(2, led\_1a, LED\_1A\_SIG);

TaskMgr.addWaitSignal(3, led\_1b, LED\_1B\_SIG, LED\_1B\_TIMEOUT);

TaskMgr.add(4, button\_2);

TaskMgr.addWaitMessage(5, messenger);

TaskMgr.addWaitDelay(6, ticker, TICKER\_DELAY);

}

void button\_1() {

static bool button\_pressed = false;

if(digitalRead(SWITCH\_1\_PORT)==LOW && !button\_pressed) {

// went from unpressed to pressed

TaskMgr.sendSignal(LED\_1A\_SIG);

TaskMgr.sendSignal(LED\_1B\_SIG);

button\_pressed = true;

TaskMgr.yieldDelay(50); // debounce

} else if(digitalRead(SWITCH\_1\_PORT)==HIGH && button\_pressed) {

// went from pressed to unpressed

button\_pressed = false;

TaskMgr.yieldDelay(50); // debounce

} // else no change so don't do anything

}

void button\_2() {

static bool button\_pressed = false;

if(digitalRead(SWITCH\_2\_PORT)==LOW && !button\_pressed) {

// went from unpressed to pressed

// select a message...

// (if we've hit the end then go back to the start)

if(theMessages[nextMessage][0]=='\0') nextMessage = 0;

TaskMgr.sendMessage(5, theMessages[nextMessage]);

nextMessage++;

// process the button

button\_pressed = true;

TaskMgr.yieldDelay(50); // debounce

} else if(digitalRead(SWITCH\_2\_PORT)==HIGH && button\_pressed) {

// went from pressed to unpressed

button\_pressed = false;

TaskMgr.yieldDelay(50); // debounce

} // else no change so don't do anything

}

void led\_1a() {

// just invert the LED and reschedule

static int ledState = LOW;

ledState = (ledState==LOW) ? HIGH : LOW;

digitalWrite(LED\_1A\_PORT, ledState);

// reschedule

TaskMgr.yieldForSignal(LED\_1A\_SIG);

}

void led\_1b() {

// led\_1b needs to process both signals and timeouts

// Both invert the LED

// Timeout increments a counter as well

static int ledState = LOW;

if(TaskMgr.timedOut()) {

timeoutCounter++;

}

// now invert the LED

ledState = (ledState==LOW) ? HIGH : LOW;

digitalWrite(LED\_1B\_PORT, ledState);

// reschedule

TaskMgr.yieldForSignal(LED\_1B\_SIG, LED\_1B\_TIMEOUT);

}

void messenger() {

// Print the passed message

char\* myMessage;

myMessage = (char\*)TaskMgr.getMessage();

Serial.print("Messenger says: ");

Serial.println(myMessage);

// reschedule

TaskMgr.yieldForMessage();

}

void ticker() {

// Print the number of timeouts and the current time

char number[10]; // for conversion of counter to a printable thing

Serial.print("Ticker says: ");

Serial.print(itoa(timeoutCounter, number, 10));

Serial.print(" timeouts on LED\_1b. Current time is ");

Serial.println(millis());

// reschedule

TaskMgr.yieldDelay(TICKER\_DELAY);

}

Some notes on the program:

1. The routines button\_1(), button\_2(), led\_1a(), and led\_1b() all use static local variables. This allows them to cleanly manage data they need to maintain between calls without creating a lot of global variables that aren't used globally.
2. The debounce/process code for the buttons is slightly different than used in earlier programs. Both work.
3. Each task has to explicitly reschedule itself with appropriate yields (yieldSignal() etc.).
4. The led\_1b() task is very similar to the led\_1a() task. The only difference is that led\_1b() has to handle a potential timeout.
5. The program also shows how messages are truncated if they are too long.
6. It introduces TaskMgr.addWaitDelay(ms). This is similar to addWaitSignal() and addWaitMessage().

**Final Notes**

In the current implementation, only one signal or message is retained for each task. If two tasks t1 and t2 send a signal or message to a third task t3, t3 will retain only the last sent one.

A task can only wait for a signal or a message, but not both. A task will receive a signal only if it had been waiting for one; likewise for messages. If a signal is sent and no task is waiting for it, the signal is discarded. If a message is sent to a task that is not waiting for a message, it is discarded.

If a task has executed a yieldForSignal() or yieldForMessage() with a timeout and the signal/message is received but the task not activated until after the timeout period (perhaps because some other task delayed everything), then the signal/message will take priority. The routine timedOut() will return false indicating that a response had been received. The routine timedOut() only returns true when the routine has been activated but no signal/message has been received.

The addWaitMessage() and addWaitSignal() routines add tasks as if they had already been set up with yieldForMessag() and yieldForSignal(). Upon completion of the appropriate processing, a task will need to reschedule itself with yieldForMessage() or yieldForSignal() if it wants to wait again for that message or signal.

More Ways to Add Tasks

Thursday, July 23, 2015

9:01 AM

This section describes a few additional variations to TaskManager.add\*(). I wrote these because, like all good programmers, I am lazy. Later, we discovered that one had a subtle variation that proved useful.

In the Signals and Messages section, we saw what is actually fairly common code: A task is created with

TaskMgr.addWaitSignal(2, led\_1a, LED\_1A\_SIG);

and when it runs, it processes a signal and then re-waits for the signal:

void led\_1a() {

// just invert the LED and reschedule

static int ledState = LOW;

ledState = (ledState==LOW) ? HIGH : LOW;

digitalWrite(LED\_1A\_PORT, ledState);

// reschedule

TaskMgr.yieldForSignal(LED\_1A\_SIG);

}

The simpler approach is to combine the two. Let the definition of the task tell the scheduler that it AUTOMATICALLY wants to wait for the signal again after it is done. Then, the code creation becomes

TaskMgr.addAutoWaitSignal(2, led\_1a, LED\_1A\_SIG);

and the routine is simplified to

void led\_1a() {

// just invert the LED

// This routine automatically reschedules to wait for a signal.

static int ledState = LOW;

ledState = (ledState==LOW) ? HIGH : LOW;

digitalWrite(LED\_1A\_PORT, ledState);

}

TaskMgr.addAutoWaitSignal() has the same parameters as TaskMgr.addWaitSignal(). Internally, it just informs the scheduler that, whenever the task finishes normally (via return; or falling through the bottom), it will be automatically rescheduled in the same manner.

Here are the new forms of TaskMgr.addAutoWait\*().

* addAutoWaitDelay(byte taskId, void (\*task)(), unsigned long int delayMs, bool startDelayed=false)
* addAutoWaitSignal(byte taskId, void (\*task)(), byte sigNum, unsigned long int timeout=0, bool startWaiting=true)
* addAutoWaitMessage(byte taskId, void (\*task)(), unsigned long int timeout=0, bool startWaiting=true)

Here are guidelines on using them:

1. The last parameter can be used if you want to have the routine start in a non-waiting state. This is very rare. If you leave the parameter off, TaskManager does what you would expect it to do.
2. You should *always* exit the task by using return; or letting it fall out of the bottom. You should (almost) *never* use a TaskMgr.yield\*(). Note that if you use a yield\*(), this will OVERRIDE the automatic rescheduling. Instead, it will do whatever form of yiielding you have requested. However, the next time you exit the task with a normal return, it will go back to rescheduling itself.

**The Difference between yieldDelay() and AutoDelay()**

There is a very subtle difference between using add() and yieldDelay(), as opposed to addAutoWaitDelay(). In a nutshell, yieldDelay() will count the delay period from the time the task is exiting, while addAutoWaitDelay() will count the delay period from when the task starts.

Consider the following two programs

void task1();

void setup() {

TaskMgr.add(1, task1);

}

void task1() {

// do something that takes 500ms

TaskMgr.yieldDelay(1000);

}

And

void task2();

void setup() {

TaskMgr.addAutoWaitDelay(2, task2, 1000);

}

void task2() {

// do something that takes 500ms

}

The first task will start at some time (for example, when the system clock reads 200ms) and run for 500ms. When it exits, the system clock will read 700ms, and it will be scheduled for 1000ms later. So the next time it executes, the clock will read 1700ms.

The second task will start at some time (for example, when the clock reads the same 200ms) and runs for 500ms. When it exits, the system clock will read 700ms. However, the AutoWaitDelay will reschedule the task 1000ms after the START of the task. So the next time it executes, the clock will read 1200ms.

The distinction is as follows:

* yieldDelay() is used when you want to delay a specific amount of time after an activity has been completed, to allow some external event to take place (for example, switch debouncing).
* addAutoWaitDelay() is used to guarantee (as much as can be guaranteed) that the task will execute on a regular and predictable basis, regardless of how long the task (or other tasks) take. Note that it is not a guarantee in that it will not be pushed ahead of other tasks that are waiting to run. It will just be available to run on a defined, periodic basis.

Networking

Sunday, July 19, 2015

9:18 AM

Basics

The networking component of TaskManager allows tasks running on different Arduino systems to interact in the same manner as they would have if they were running on the same system. It uses RF24 modules and the RF24 library to establish communication.

I recommend using the tmrh20 RF24 library, located at <http://tmrh20.github.io/RF24/> . It is compatible with the base ManiacBug library, and has enhancements that improve its compatibility with a wider range of Arduino modules.

Basic wiring of the RF24 is not covered here.

There are two components to establishing a TaskManager network:

1. Wiring the Arduinos with RF24 modules.
2. Establishing communication within the program.
3. Sending messages between tasks.

Establishing Communication

You connect an RF24 module to an Arduino by attaching eight wires: 3.3V, GND, MOSI, MISO, SCK, CE, CSN, and INT.

* 3.3V, GND, MOSI, MISO, and SCK are all standard lines on each Arduino. MOSI, MISO, and SCK will be shared with any other SPI devices you may attach.
* INT is optional, and the current implementation of TaskManager does not use it. As such, you do not need to connect INT.
* CE and CSN need to be connected to distinct digital pins. For the purpose of the rest of this section, we will use 9 for CE and 10 for CSN.

Once this is done,

TaskMgr.beginRadio(byte nodeId, byte CE, byte CSN)

Is used to start radio operations. After beginRadio() has been called, the following happens:

* The byte value nodeId gives this node an addressable name in the range 1..254. Do not name a node as node 0 or node 255.
* This node is given the name "xTMGR", where x is replaced by the byte nodeId. Note that unless the byte nodeId is a printable character, the node name will not be printable.
* A receiver task (taskId in the "system" task range of 192-255) is started. It is a polled task, and will process incoming radio packets directed at this node.

Sending Messages and Signals

There are two basic methods of sending messages and signals to a task running on the local node:

* sendSignal(byte sigNum)
* sendSignalAll(byte sigNum)
* sendMessage(byte taskId, void\* buf, int len)
* sendMessage(byte taskId, char\* message)

Each of these has a variant that sends the message or signal to a task running on a remote node:

* sendSignal(byte nodeId, byte sigNum)
* sendSignalAll(byte nodeId, byte sigNum)
* sendMessage(byte nodeId, byte taskId, void\* buf, int len)
* sendMessage(byte nodeId, byte taskId, char\* message)

Using a nodeId value of 0 will cause the message/signal to be sent to the local node.

A Quick Example

Final Notes

We have not experimented with several SPI devices on the same Arduino. However, the following should be noted:

1. All devices will share MOSI, MISO, and SCK.
2. Each device will need its own CE and CSN digital lines.
3. The standard SPI interface has routines (beginTransaction(), endTransaction()) and an associated SPISettings class that are used to preserve SPI driver settings across objects that use different SPI settings sets. The tmrh20 RF24 library makes use of these routines to preserve its local integrity. However, other libraries implemented on the SPI bus may not use this interface extension. If other devices are used and the libraries are not compatible, beginTransaction(), endTransaction(), and SPISettings will need to be managed manually.

Programmer's Summary

Monday, July 13, 2015

5:10 PM

The full API to this point

TaskMgr.add\*

Void add(byte id, void (\*task)());

Void addWaitDelay(byte id, void (\*task)(), unsigned int msDelay);

Void addWaitUntil(byte id, void (\*task)(), unsigned long msUntil);

Void addWaitSignal(byte id, void (\*task)(), byte sigId, unsigned int timeoutMs=0);

Void addWaitMessage(byte id, void (\*task)(), unsigned int timeoutMs=0);

Void addAutoWaitDelay(byte id, void (\*task)(), unsigned int msDelay, bool startDelayed=false);

Void addAutoWaitSignal(byte id, void (\*task)(), byte sigId, unsigned int timeoutMs=0, bool startWaiting=false);

Void addAutoWaitMessage(byte id, void (\*task)(), unsigned int timeoutMs=0, bool startWaiting=false);

TasMgr.yield\*

Void yield();

Void yieldDelay(int delayMs);

Void yieldUntil(unsigned long int ms);

Void yieldForSignal(byte sigId, unsigned int timeoutMs=0);

Void yieldForMessage(unsigned int timeoutMs=0);

Message and Signal

Bool timedOut();

Void\* getMessage();

Byte myId();

TaskMgr.send\*

Void sendSignal(byte sigId);

Void sendSignalAll(byte sigId);

Void sendMessage(byte taskId, char\* msg);

Void sendMessage(byte taskid, void\* data, int dataLen);

Suspend, resume, kill

// need to define

Void suspend();

Void suspend(byte id);

Void resume(byte id);

Misc

Unsigned long runtime();

Size\_t printTo(Print& p);

How TaskManager Works

Sunday, July 19, 2015

8:40 PM

This is a quick summary of how TaskManager works. In essence, a peek under the hood.

You don't have to read it now. You can read it later, if you want to. Or you can skip this entirely. But it might be fun to look.

Note: Within this section, yield\*() refers to any member of the yield function family, add\*(), refers to any member of the add function family, etc.

From the user's perspective, a task is a procedure that runs and then exits (by either returning or yielding). TaskManager takes this procedure and bundles it into an element containing

* The user's procedure
* Its ID (a value between 1 and 192 (for user tasks) or between 193 and 255 (for system tasks)
* Its current status (available to run, waiting until an amount of time has passed, waiting for a signal, etc.)
* If it is waiting for a time, the clock time that it is waiting for
* If it is waiting for a signal, the signalID it is waiting for
* A message, if it had been waiting for a message
* If it should auto-reschedule after it exits
* Other information as needed

All known tasks are bundled into a *ring*. A *ring* is a structure like a linked list. It is a group of objects set up so that each object contains

* Some data (in this case, the abovementioned task information)
* A pointer to the next element in the ring
* A pointer to the previous element in the ring.

The last points to the first and the first points to the last. The ring has no beginning or end, only the concept of what the current element is.

Here is a diagram of a ring of three tasks. Note that there is a fourth, the "null" task. The "null" task, whose taskID is 255, does nothing. Its purpose is to ensure that the ring is never empty, and there is always something to do.

loop\_led\_1 cur

null loop\_led\_2

loop\_led\_3

This ring allows TaskMgr to go through its tasks. It runs the "current" task. Then when the "current" task is done, it moves to the next. Then it runs that new "current" task.

Each task (as mentioned earlier) has a variety of associated information. It has a set of bit flags telling if it is active, if it is waiting for a message, if it is waiting for a signal, or if it is waiting for a certain time to pass. Additional information tells what signal it is waiting for, what time it is waiting for, and so on.

The TaskMgr's loop() operation is the component that ties everything together. It performs the following:

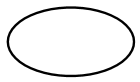
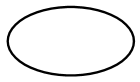
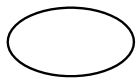
1. It looks for the next available task. An available task must be active, must not be waiting for a signal or message, and if it is waiting for a time to pass, that time must have passed. (Note that the null task is always available to run.)
2. It then calls the task. It calls it using C's setjmp/longjmp functions. This is old-style C, but it allows, for example, a task to yield\*() from anywhere, including from within procedures the task may call. The yield\*() can also pass information back to TaskMgr, including what kind of yield is being used.
3. When the user's procedure is called, the information from the yield\*() or the return is used.
   1. If it is a yield for time, signal, or message, the current task is then marked as waiting for the particular information.
   2. If it is an ordinary return or ordinary yield(), and if the task is an auto-reschedule task, it will automatically reschedule itself to re-wait for the signal, message, or time to pass.
   3. Otherwise, the task is left as active.
4. The TaskMgr's loop() then exits. The next time it is called (it is "loop()", after all!), it will pick up where it left off, and find the next available task.

From here, the operations are fairly straightforward.

* add\*() adds the supplied task to the end of the task list (that is, right before the current task). The new task's flags are set as needed.
* signal\*() and message\*() will find appropriate targets and pass the information to the target.
* yield\*() saves information (as needed) and then executes a longjmp() to get back to the TaskMgr's loop().
* Other routines are left as an exercise to the source-reader.

A few final notes:

* Yield\*() never returns. The next time through, execution starts at the top. Techniques to get around this are discussed in the TaskManager Techniques section, and the section State Machines goes into greater depth.

C:\67A35465\88032A2A-C16E-4C60-9F70-E0464B428E3B_files\image007.pngC:\67A35465\88032A2A-C16E-4C60-9F70-E0464B428E3B_files\image008.pngC:\67A35465\88032A2A-C16E-4C60-9F70-E0464B428E3B_files\image009.pngC:\67A35465\88032A2A-C16E-4C60-9F70-E0464B428E3B_files\image013.png

System Impact

Saturday, July 25, 2015

4:13 PM

This is a collection of random notes related to performance of the TaskManager library.

* The task switch time, as measured on a 16MHz Nano, is around 20 microseconds. That is, it takes around 20us to switch from one task to the next. This assumes that no tasks were skipped while looking for the "next" task. It would skip tasks if the skipped tasks were waiting for messages or in a delay state, for example.
* We would expect that it takes less than 20us to go past each switched task. This hasn't been measured.
* Memory use of TaskMgr object
* Memory use of each task that has been added
* Tasks are run in the order they were added.

Future Plans

Tuesday, November 10, 2015

10:53 PM

Just a few notes for future plans.

Sending Messages and Signals Everywhere

We will add the ability to send messages to every node the current node can find. This will be done through the following changes:

1. There will be a new method, TaskManager::FindNodes(). This routine will poll and find all of the nodes that it can. This will test every node between 1 and 254, so it may take some time to complete. It will only need to be called once, during setup().
2. Sending to node 255 will cause the information to be sent to every node found by FindNodes().

Getting a Node's Status

We will add the ability to poll a node to see what its current status is. This will return a NodeStatus object with a series of fields. One field will be a boolean, m\_exists, that tells if the node exists. Other fields will be defined in the future.

Getting a Task's Status

We will add the ability to inquire about the status of a task on this or another node. This will return a TaskStatus object with a series of fields. One field will be a boolean, m\_exists, that tells if the task exists. Other fields will be defined in the future.

This routine will not be able to be called on node 255 ("all nodes").

Suspend, Resume, and Kill

Performing operations on tasks, both locally and on the