An Introduction To Simple Scheduling

(Primarily targeted at Arduino Platform)

"I'm late
I'm late
For a very important date.
No time to say "Hello, Goodbye".
I'm late, I'm late, I'm late."
(White Rabbit in Disney's Alice in Wonderland)

Time – the keeping of time, performing software actions at correct time, not taking too long to perform software tasks. While allowing other actions happen when some part of the software will wait for a long time.

These are all things those programming from students, novices, hobbyists to professionals have trouble with to a lesser or greater extent. Grasping the concept that software or a hardware action takes time unrelated to anything else that can be seen, is a hard one for most people.

As some things take a lot of time to perform, often software is written to buffer up data and for the buffer to be read later, using interrupt routines, and the other way round data collected by interrupt routines and buffered to be read by software when it is available. The classic examples are serial ports on computer systems, which are generally buffered in this way so your programme does not have to spend all its time checking to see if the hardware is ready to send or has received data.

This is due to the much slower speed of serial ports for example the time it takes to send one byte at 9600 baud is approximately 1 milli-second. Compare this with a 12 MHz clocked processor, which then has a clock cycle of 83.3 micro-seconds. If we were to assume an average of 4 clock cycles to perform one machine code instruction, then in the time it takes to send one byte via the serial port we could be performing 3,000 instructions of our programme. Similarly when using computers to control things (even switches and LEDs), there is lots of time between an event like a button push, when we don't need to be waiting in software for the button press commands. We have **spare** time so we could be doing something else. Just like when the adverts appear on television, we do not have to watch them we can go make a cup of tea and come back to the programme after the adverts.

With larger sizes of data and speeds for things like hard disk drives and networks, other forms of hardware transfer directly into buffers are used, generally Direct Memory Access (DMA), for chunks of data referred to as blocks or packets. These are not covered here, but are often used.

So what do we do to achieve better use of time?

Well this depends on the type of application and its requirements for what must be done in what timescales. If we are just monitoring room temperature and turning heaters or fans on or off, timescales are in **computing terms**, very loose. By this what is meant is that if the temperature is not read at exactly once every half second but half second +/- 10 milli-seconds it would not matter as the heaters and fans will take longer to start or stop than that. However if we are using a computer to control the sparking of a car engine, we must always be ready to action the necessary hardware usually in the order of tens of micro-seconds. So we would need in **computing terms**, very tight scheduling of our programme and what time was used for what. Often in these cases there are various pieces of specialised hardware peripherals to help us perform these actions.

In order to achieve this we need to break down the software into sections that can be executed in part or full many times a second or not at all if not needed. Just like you should modularise your software into functions, and often separate files with several functions in, you need to split the software into blocks of code that can be executed when required. This may be a single function or a function that calls many other functions. What you will be doing is splitting the software into a collection of tasks to do. When and how often depends on many factors about the application you are creating.

You need to work out

How each task or thing has to be done?
How long each task is likely to take (if one is very long can it be split)?
What are the long time periods where the delays are hardware delays (serial, LCD...)?
What tasks are not running all the time but commanded (by switch or command)?
If all tasks are running how long will it take to perform all of them all once?

Word of caution, often I hear people start to look at this and want **every task** to be run as often as possible, and because computers are fast and each task must be started at 1 micro-second intervals. Well nice as that may be in **most** computers and with a lot of tasks, this is a waste of time and impractical. Lots of things being controlled, read, written to or even calculated have **real world** limits, the biggest of which is **human beings**! Take for example you want to monitor a variable resistor on an analogue channel, and use that to change the brightness of a display or the temperature that a heater must reach; the issues are often **human beings** will not notice the difference between taking readings 10,000 times a second and 100 times a second, also for things like heaters they take a long time to get up to temperature or cool down.

So recognise the real world limits, like how many buttons could be pressed a second, if I buffer data from a PC or elsewhere how often do I really need to check if a complete message is there.

When dealing with slow devices like serial ports and LCDs, make the software, make the tasks execute as much as they can and then leave to finish later. An example would be a 20 x 4 LCD text display, I have measured that to write one line of that type of display can with some software take 3 to 5 ms to write ONE line, (with tweaks I have managed to get most down to 1 ms). This is a **long** time in computing terms, so if you expect to run another task after the one updating the LCD at 1 ms afterwards, it **will** be late in running by 2 to 4 ms. This action is called **blocking** – where one task takes control and does not relinquish time to any other task, possibly causing other knock on issues.

So tasks have to be as non-blocking as possible and **not** use delay functions or similar, but relinquish control to other tasks and carry on the next time the task is called. On Arduino functions like delay(), pulsein() and Print() are **blocking**, these are not the only ones, but the major ones people use. All of the software has to be designed to be non-blocking, or have methods to allow a few blocking tasks only when it is wise to do so (like turn heaters and motors off before doing a long debug dump to PC).

Run to Completion (Blocking)

This action of blocking is very common in computer systems even your PC/Tablet/Phone. At some point in time many things have to be blocking to ensure that certain things happen at right time and completely.

However you need to **design** your whole application to **reduce** the length of time blocking occurs, you cannot always avoid it, just reduce it to a minimum.

Your tasks are normally first written as run to completion for example -

- Write a line of characters to LCD
- Energise an ultrasonic transducer and wait for the echo to come back to calculate distance
- Start and wait for an ADC conversion, change the value into units you want
- Worst of all sit and wait for a user to push a button or for a command from a serial port.

'Run to Completion' means each function or task or section of code expects (barring interrupts) to run till it has done everything in that section of code. There is no mechanism in standard environments like Arduino and many small computers, to force the task to suspend and come back later to complete. This may seem a problem but by carefully splitting your code into sections and knowing what sort of time that section will take means you can schedule operations for better efficiency and overall response time.

So if you are waiting for a button press, check if it has been pressed if not, exit and come back later, let something else run. Only execute the main parts when something to do. Some things within environments like Arduino will need helper functions or doing a different way, these can vary depending on your application and you will need to design for **your** application what works best, there is no one size fits all for this, each application is different with different requirements.

Co-operative Scheduling

When we have a list of tasks we have to organise in what order and how often they run, the simplest of these methods is co-operative scheduling, as other types of scheduling requires more resources (programme and data space) and more complications in how to work. The more complicated types of scheduling are more suited to larger computers with programmes that are run or not depending on what users' commands like on your PC when you choose to browse internet, edit a document or look at your email.

Sometimes people refer to co-operative scheduling as Round-Robin, well this is partly true as all schedulers have a list they repeatedly go 'round' to check what task they will do next , but they may change order or restart the list at any time.

Co-operative scheduling is the easiest to implement with Run to Completion tasks, as the main scheduling loop then consists of a table of tasks that the scheduler goes round repeatedly in order call the tasks as necessary.

How the tasks are called depends on complexity of the scheduler, but a minimum you require for each task in the scheduler list is -

- Task address (the address of top function in a task)
- When to run – some kind of count related to time to say when to execute next
- Delay to execution – what value to add to When to run (or current time) to run next time, sometimes a value of 0 is used to denote do not run this task until this value changes.

Often a scheduler will perform the task of adding this delay after the task has completed so that this is done in one place and avoids duplication and potential errors in code.

- Status (Task enabled/disabled) – you cannot always remove tasks from lists on all systems, having the ability to leave a task in an idle state until something else has completed is useful. So if a task has completed it can disable the task from running again, until some other event starts it again.

Example

The example we will use here has various activities, some of which you may use and some are for demonstration of tasks that take different amounts of time -

1. Read a pot and adjust a PWM output (just like brightness control), ten times a second
2. Flash an LED at 4Hz (4 times a second ON and 4 times a second OFF)
3. See if a button has been pressed – if so start flashing a second LED at 10Hz for 2 seconds
4. Checksum the area of RAM and save the result, every pass.
5. See if a second button has been pressed and if so output the checksum to LCD
6. If either of two other buttons have been pressed send to serial the statistics or current scheduler activity

Now you might think that (5) should just be done by interrupt routine, but that would make that interrupt routine **block** the whole system more (orders of 1 to 5 ms see above). During that long time in interrupts, many other interrupts are being blocked and potentially causing problems. If you use interrupts on switches use them to set *volatile* flags that can be checked later, as a delay of 10 to 20 ms before the data is displayed on LCD will not be visible to the **human user**.

Task 4 in the above is to simulate a long running task that has to be done regularly, and how it affects things.

Task 3 is a task to show it is up to the task to determine how often it is run and when it has finished. It is not to job of the scheduler to do this, it should allocate when tasks are to run, and stopping tasks is a separate activity. Even on your PC/Tablet you have to use special utilities to do this.

Tasks 5 and 6 are related to the scheduler method implemented as it has the ability to collect data as it is being run so you can analyse if things are working or not, this partly relies on serial port speed being 115,200 and large buffers for serial for it not to block operations and delay the next task execution time.

Reviewing the task list you will see we the following types of task

1. Continuous runs every pass through the scheduler list
2. Intermittent tasks that every so often but not every pass of scheduler list
3. On demand tasks that respond to events and run once or several times then stop.

This is where the items described in previous section come into play, so we can disable or enable tasks and set when to run, as in this table –

Type	Delay to Execute time	
	When Enabled	**Disabled**
Continuous	Minimum loop time	Never Disabled
Intermittent	Time delay required	Never disabled
On Demand	Time delay required	0

Note “Minimum loop time” is the smallest time increment the scheduling loop allows so at next read of the list, it will be executed.

Run Time or Compile Time Task allocation

The choice of how to build your scheduler list mainly depends on the system it is being run on and its capabilities, **then** your application requirements.

Whatever method you use, the address of the top function of your task has to be made available to create the scheduler task list. When compiling in environments like Arduino, **ALL** tasks are known at compile time and cannot change, where as on a PC you load more and more programmes to run. On smaller systems you often do not have the ability to load extra programmes, so creating extra code to build a scheduler task list as part of the running code time is expensive on code and **time**.

This factor is especially true of AVR based Arduinos (like Uno and Mega), as this uses a Harvard architecture processor, with limited amounts of RAM you can use for Data only. There is **NO programme space RAM** to load programmes into. These sizes of processors often have limited RAM so loading extra programmes is difficult, and environments like Arduino do not have the mechanisms for this (without a lot of work by you).

Even on ARM/SAM based boards like the Due, there is no standard way of having your application loading a programme from SD card or USB device, then running it and potentially closing it. This would require a lot of work and potentially a lot of RAM being used.

Implementation of Example

So we have our basic tasks, as a reminder –

1. Read a pot and adjust a PWM output, ten times a second
2. Flash an LED at 4Hz (4 times a second ON and 4 times a second OFF)
3. See if a button has been pressed – if so start flashing a second LED at 10Hz for 2 seconds
4. Checksum the first 1k of RAM and save the result, every pass.
5. See if a second button has been pressed and if so output the checksum to LCD
6. If either of two other buttons have been pressed send to serial the statistics or current scheduler activity

So let us give the tasks some names and put them into a list as code -

```
int ( * const tasks[])( int, int ) =
                    {
                    brightnessCheck,
                    LED4hz,          // Flash LED 1 at 4 Hz  (continuous)
                    LED10Hz,         // Flash LED 2 at 10Hz for 2 Seconds
                                     // on switch press. Initially off
                    CheckRAM,        // Checksum 1st 1k of RAM every 10ms
                    CheckLCD,        // output Checksum to LCD
                    statisticsCheck  // See if we output statistics
```

```
                    };
```

The list is in the form of an initialised array (at compile time), the important things to note –

- Array is called ‘tasks’
- Contents are names of functions of type determined by array
- Array is a constant array containing constant pointers to functions so they reside in Flash memory, so does this array.
- The functions all take TWO integer parameters (explained later)
- The functions return ONE integer (explained later)

Now we need to create a similar list of our data to determine when to run each task, but we have a collection of data about each task, which could be implemented as a class or a structure, here we will choose the structure to be

```
struct TaskList {
                unsigned long next; // next execution time in ms
                unsigned long last; // last execution time in us
                int status;         // current task status 0 stopped,
                                    // -ve stopped with error,
                                    // 1 start,
                                    // >1 user status (and active)
                int interval;       // interval between starts in ms
                int executed;       // did run this pass
                };
```

To use this structure we add the following

```
#define _MAX_TASKS  (sizeof(tasks)/sizeof(const int(* const)()))

// Array of task details
struct TaskList taskTable[ _MAX_TASKS ];
```

The define tells us at compile time how many tasks there are, so we can then define our array called taskTable with the **matching** number of entries, each being a structure of type TaskList. This is the list the scheduling loop will read and process every scheduling loop, and when it needs to execute a task call the function at same index in our previous array ‘tasks’.

Each element of the taskTable array has a variable for –

- When to execute (next),
- Status = 0 stopped, greater than 0 running, less than 0 stopped in error
- How often to execute (interval)
- If run this pass (executed), needed to understand logging output
- How long it executed when last run (last)

The last is useful for use to measure how many microseconds a task took to run, so later we can actually **measure** how long any task took, to solve problems or flag errors.

TaskCreation

Each task is a top level function (that may call lower level functions), to perform a specific task. The basic task function structure is –

```
int task( int ID, int status )
{
// do something here
return value;
```

}

The parameters passed in are the task's `ID` (index in the arrays) and current `status`. The first allows us to store this `ID` elsewhere so we can use it for helper functions and initialising the task in the scheduler. Where as `status` is the current task status, which will be updated when we exit by the line `return value;`. This way we can use the task status as our task's state machine variable. So when we call the task with status = 0 we can initialise the task and anything it has to do with GPIO etc., then return with what the next state is for next time called. This way we can yield to other tasks and carry on what we were doing next time.

The easiest way in most languages especially C and C++ to implement a state machine is to use a `switch` statement and use each `case` statement as a particular state. For example –

```
int task( int ID, int status )
{
switch( status )
      {
      case 0:     // Initialise task
                  // Do initialisation actions
                  setInterval( ID, 100 );
                  status = 1;
                  break;
      case 1:     // Start task
                  // perform actions to prepare for running of task
                  status = 2;
                  break;
      case 2:     // Main iterations
                  // Do looping actions
                  If( nextpass == last )
                        Status = 3
                  else
                        status = 2;
                  break;
      case 3:     // Last iteration of task
                  // Do last iteration actions
                  status = 0;
      }
return status;
}
```

So examining each 'state' of our state machine we have

State 0 Set task interval of 100 ms to call again, with state of 1

State 1 Prepare for running many times

State 2 Perform the bulk of task duties
If more than one more time to do set next state to 2 (this one)
Else set state to 3 (last pass state)

State 3 Perform last iteration actions
Set state to 0 for STOPPED, to await starting again.

Note `setInterval( )` **MUST** be called in state (case) 0, to ensure we are called again, but only this task knows how often it should be called. However you may change the interval as required in any of the other states, but setting the time to zero is **not** recommended.

So below is a task to flash an LED at 4Hz, so that is 8 calls, 4 to turn ON and 4 to turn OFF

```
int LED4hz( int ID, int status )
{
switch( status )
  {
  case 0: // initialise
          setInterval( ID, 125 );
          status = 2;
          break;
  case 1: // Start
  case 2: // write LED on
          LED.Toggle( STATUS_0_W, 0, 0 );
          status = 2;
  }
return status;
}
```

Note how small the function is the bulk of the work of when to do this is handled by the scheduler, and it is the same software for all tasks, so reduces the code overhead and complexity.

If you look at the code you will notice that the states 1 and 2 (`case 1`: and `case 2:`) are labels together with nothing between, a `switch` statement runs a state (`case`) until it reaches a `break` statement before exiting, so we can make several states do the same thing. By adding the state 1, we are stopping inadvertent starts of task causing problems and ensuring we stay in state 2 after.

LED.Toggle is a function I use to toggle i.e. reverse the state of particular LEDs, but this function could change up to 24 at a time. This is using lower level helper functions to assist in your tasks and also make each task smaller and easier to read or maintain.

Now we have the basics of tasks scheduling and creation for a compile time structure covered.

Now what to do when it does not work

The prime reasons applications fail when using a scheduler is not understanding how long things take and interactions. Such as –

1. Task Loop Interval too small – basically the scheduler is expected to run through its list of tasks **too often**. So if all tasks run, this takes too long, so **most** if not **all** next schedule times are late before we finished going through the loop.
2. One or more tasks is too long – Some tasks are not realised to take a long time, like sending 1000 characters over serial at 9600 baud will take **1.04 seconds**, so unless you have serial buffers that can handle that much data, your task will sit **blocking** waiting to send the data. Increasing the baud rate will reduce the blocking but rarely will it stop it as serial ports do not operates ate 10 GH\z.
   Better to get the task to send almost a buffer full of data, and come back later to send the next section until finished. Use your state machine principles to break it down, something like –
   a. State 1 Print header row
   b. State 2 print a line at a time
   c. State 3 finish output and any extra blank lines or totals
3. One task produces wrong results – often people set tasks to do separate sections of processing information like one task to read a position, another task to drive motors based on position. Make sure the **order** of tasks is correct and the interval on each one matches what is possible. Remember the time lags and real world limits of how things work.

4. System is too slow, often you may have fast tasks, but because you have one slow one set the Task Loop Interval too long so no task is overdue. However that could mean that because the one slow task is infrequently used and does not affect anything else, you could reduce the Task loop Interval and accept that when that task is run all others **will** be late (overdue) on next loop pass but will then get back in sync. Alternatively find a way to get that task to operate in smaller chunks of time.

   Obviously if you have one long task like send lots of serial data and you must block all other tasks during that, make sure the other tasks are in a **SAFE STATE**. So if you have a motor or heater running you stop them **FIRST** before doing the long task. You do not want things catching fire because you can no longer monitor the temperature of a heater to turn it off at temperature.
5. **Lack of measurements –** too often I see people make wild guesses at what is going on without, monitoring with test equipment or using software to tell you what is going wrong. Such things as what is the longest time a task has run for, what was the longest loop time. Then proceed to make changes (often several at a time) and may get things working for the **WRONG** reason, so the next minor change breaks it again.

MEASUREMENTS AND DATA ARE THE MEANS TO DIAGNOSIS

To assist with especially the last item the scheduler used in the example has the ability to takes various logging and statistical information, which can be read by your software or as in the example sent to serial port. These details included

Logs
- Was task executed this task loop iteration
- What was LONGEST execution time of this task
- Task's current status

Statistics
- Task schedule loop, start and finish time
- Maximum schedule loop time
- This task loop overdue by how many ms
- Running Average of overdue times
- Maximum Overdue time
- Longest task execution time and which task

Note every time the Statistics are requested the items that are longest and maximum are reset to zero so new longest and maximums can be seen over time. See example Arduino sketch for viewing on serial monitor at 115,200 baud.