Dispatching and Monitoring Tasks from loop()

Table of Contents

1 Introduction

This document describes a standard approach to executing tasks from the loop() function. Following this approach has these advantages:

- enables automatic monitoring of the tasks CPU utilization and the dispatch latency
- allows prioritization of tasks that need to be serviced quickly after timer expiry
- enforces execution of a single task per loop() cycle, which reduces priority task latency
- allows use of standard macros to reduce the effort to install tasks

There are two types of tasks that can be dispatched:

Timer Tasks are controlled by a timer which is a time stamp expressed in millis(). When the system time advances to the point where it is equal to or greater than the scheduled value for the task, the task is executed. Measurements are captured before and after the execution which are later used to calculate CPU performance measurements for the task.

Condition Tasks are controlled by a logical condition that can be used in an IF statement. If this statement is true, the task is executed, again with capture of measurements which are used to calculate the task's CPU performance.

It is very important that the code for either type of task handles resetting or disabling the timer or condition that caused it to be called. If this is not done, the task will be dispatched repeatedly on every iteration of loop().

For timer tasks, this is usually done by setting the timer scheduling control variable to the current time plus the interval between task executions. If a timer task is to be disabled, the control timer is set to 2,000,000,000 which actually schedules it for 25 days after robot startup.

For condition tasks, this is done by modifying the variables involved in the condition. A common approach is to set a "request" bit in a flag word to trigger task execution. The task would clear this bit so the task will execute again only if it is re-requested.

The detailed instructions for task installation will contain more details and examples.

The procedures to install timer and condition tasks are quite similar. We'll provide one procedure description with notes identifying places where there are differences.

After describing the installation process, we'll go over the CPU measurements and how to interpret the displays that are available on either the serial monitor or MQTT health topc.

2 Installation of a New Timer Task

There are several steps to go through to install a new timer task that accomplish the following:

1. specify parameters related to the task:
    1. name of task to be run
    2. number of milliseconds between executions
    3. short form name to be embedded in variable names related to this task
    4. very short name to be used in display of CPU usage
2. Generate the variables needed to track CPU performance of this task
3. Insert code to do initialization of the timer at startup
4. Insert code in the task to be called that either resets or disables the timer
5. Insert code that runs once per second to calculate and display the task's CPU usage

We'll go over each of these in more detail, and there's a procedure summary at the end of this document.

2.1 Specify Task Parameters

These parameters will be used later in this procedure:

2.1.1 Name of Task To Be Run

This the name of the function to be called when the timer expires. It is usually a long descriptive name, in camel case, with parentheses, but no arguments and no terminating semi-colon.

Example: scanSensorReadings()

The task will be called by a statement that has this value by itself followed by a semi-colon.

2.1.2 Number of Milliseconds Between Executions

This is a decimal number corresponding to the number of milliseconds between scheduled execution of the task.

Example: 200

This would cause the task to execute every 200 milliseconds, or 5 times per second.

2.1.3 Short Form Name

This is an abbreviated version of the task name that is embedded in the variable names that are used for the various task-related parameters. A length of 4 to 8 characters is recommended, and it should follow our camel case convention.

Example: sensor

2.1.4 Very Short Name

This version of the name is used as a label to identify the CPU statistics that apply to this task. We want to be able to display CPU info for a large number of tasks, so we need to minimize the number of columns for each task so the display still fits on one line. For this reason, this name should be as short as practical, ideally 1 to 4 characters. No need to use camel case, use whatever makes it readable.

Example: Snsr

2.2 Generate the Variables

There are 8 variables for each task that control dispatching and track CPU performance. If we follow our example that uses the short form name sensor, these are:

Variable	Function
sched_sensor_mills	The millis() time that this task last was actually dispatched
next_sensor_mills	The millis() time that this task was scheduled to be dispatched
cum_sensor_delay	The cumulative delay in millis() between the time the task was scheduled for, and when it was actually dsipatched
max_sensor_delay	The largest difference in the last second between scheduled time and actual dispatch time, in millis()
lates_sensor	The number of times that the actual dispatch time was later than the scheduled dispatch time
time_sensor_mics	The cumulative elapsed time for execution of this task over the last second, in microseconds
max_sensor_mics	The largest execution time for this task over the last second
period_sensor	The number of milliseconds between scheduled executions of this task

The good news is that the is a predefined macro, loopTaskVars(), that defines all these variables, given two pieces of information:

1. The shortform name of your task, sensor in our example
2. The number of milliseconds between task executions, 200 in our example

The macro call for our sensor example would be:

Example: loopTaskVars(sensor,200);

This is done in the file global_variables.cpp, and you can find the location by searching for loopTaskVars. This is also a handy place to declare your task, which also documents the full name for the task, so a typical entry looks like this:

```
loopTaskVars(sensor,200);     // check sensor readings
void scanSensorReadings();    // in deviceSupport.cpp
```

2.3 Insert Initialization Code

Before task dispatching can function, some initialization needs to be done. Code to do this is placed in the function setupTasks() which is in deviceSupport.cpp.

For timer tasks, this consists of writing an appropriate value into the time variable, consisting of the current time plus the repeat interval for the task, which is one of our standard variables:

```
next_sensor_mills = millis() + period_sensor_mills;
```

For condition tasks, the condition is usually disabled until some function needs it to run, in which case that function would enable the condition.

If our example was a condition task, triggered when a request bit doSensors is set in the variable taskFlags, we might initially disable it like this:

taskFlags = taskFlags & (~ doSensors) // ~ does a 1's complement negate – reverses all bits

or, more simply, we could use a boolean variable as the task request flag, and disable it like this:

bool requestSensors = false

assuming the task dispatch looks like this:

```
conditionTask(scanSensorReadings(),sensor,requestSensors); // handle sensors
```

2.4 Put Reset Code in the Task Function

After a task has been dispatched, it needs disable the mechanism that triggered the dispatch. If this isn't done, the task will be dispatched again for every cycle of loop().

In the case of timer tasks, this is usually done by adding the scheduling interval to the timer variable. This prevents an immediate re-triggering of that task as well as scheduling it to run again at the appointed time. If for some reason the timer, and dispatching of the task, need to be disabled, this can be done by putting the value 2,000,000,000 into the timer variable. This is equivalent to scheduling the task for 25 days after the robot started up. Resetting the timer value is usually the first thing done in the task, and a typical example looks like this:

```
next_sensor_mills = millis() + period_sensor_mills;
```

For conditional tasks, the resetting process usually consists of undoing the variable setting that caused the condition to become true. Examples of this are in the previous section on initialization.

2.5 Setup the Once Per Second CPU Usage Calculation

One of the timer tasks is called oneSec, and it can be found after loop() in main.cpp. Its repeat interval, period_oneSec_mills is set to 1000 milliseconds, so it runs once per second.

It uses the boolean flag firstOneSec to detect its very first execution, and in that case, it just zeros the variables used to accumulate CPU performance data for each task. This is done by the taskProcInit(taskID) macro, and you'll need to all a call to this macro for your new task, using the short form task name, like this:

taskProcInit(sensor);

In the normal case after the initialization there are 2 blocks of macro calls that perform the calculations and display of the CPU usage information. You'll have to make a one line addition in each block.

In the first block, there is a call to a processing macro for each task to do the calculations, and append the resulting information to a report string called rep. The different types of task have different macros to do the processing, and generate slightly different report information.

For a timer task, the macro is doTimerProc(taskID,label);
For a condition task, the macro is doConditionProc(taskID,label);

where taskID is the short name for the task, sensor in our ongoing example, and
label is the very short name in quotes, which is added to the report string. “snsr” in our example.

You’ll need to add the appropriate macro call for your task to this block of macro calls.

The second block of macros is at the end of the oneSec function, and its role is to zero out the various counters to start the acumulation of data for the next second. This uses the same macro for each task as was used in the oneSec initialization:

```
taskProcInit(sensor);   // where sensor is the taskID, or short name for our task
```

2.6 The CPU Performance Display

The CPU performance information is displayed, once per second, via a trace macro at the end of the oneSec function in main.cpp. Trace output can be directed to the serial monitor, the MQTT health topic, or both by the normal method. If you have other debug information coming out on the serial monitor, it can be helpful to direct CPU performance info to MQTT to unclutter the console.

If the amount of CPU info displayed is excessive, you can remove info for tasks that aren’t of interest by commenting out the doTimerProc() or doConditionProc() macro calls for tasks that you’re not interested in.

A typical CPU info display directed to MQTT might look like this:

```
aa/hfr48B8/health M) oneSec{14}[278962]> <CPU-Info>

where aa/hfr48B8/health       is the MQTT message prefix
      M) oneSec{14}[278962]> is the trace prefix (message type, function, traceID,timestamp)
      <CPU-Info>             represents the actual CPU info, described below
```

The actual CPU info would look like this:

```
CPU: 13.27  /Oled: 0.0  0.0 0 /MQTT: 0.0  0.0 1 /webMon: 0.6  0.1 0 /Flow:12.3  0.4 /1Sec: 0.4  0.4 0 /
```

Where the fields have the following meanings:

`CPU: 13.27`	Total CPU used in the last second, as a percent
`Oled: 0.0  0.0 0`	CPU usage for the timer task with label “Oled” 0.0 Total CPU used by this task in last second, as a percent 0.0 CPU utilization of the longest call to this task in last second, as a percent 0 number of times this task was dispatched at least 1 millisecond late NOTE: timer tasks have 3 numbers, conditional tasks have 2
`MQTT: 0.0  0.0 1`	CPU usage for the timer task with label “MQTT” 0.0 Total CPU used by this task in last second, as a percent 0.0 CPU utilization of the longest call to this task in last second, as a percent 1 number of times this task was dispatched at least 1 millisecond late
`webMon: 0.6  0.1 0`	CPU usage for the timer task with label “webMon” 0.6 Total CPU used by this task in last second, as a percent 0.1 CPU utilization of the longest call to this task in last second, as a percent 0 number of times this task was dispatched at least 1 millisecond late
`Flow:12.3  0.4`	CPU usage for the CONDITION task with label “Flow” 12.3 Total CPU used by this task in last second, as a percent 0.4 CPU utilization of the longest call to this task in last second, as a percent
`1Sec: 0.4  0.4 0`	CPU usage for the timer task with label “1Sec” 0.4 Total CPU used by this task in last second, as a percent 0.4 CPU utilization of the longest call to this task in last second, as a percent 0 number of times this task was dispatched at least 1 millisecond late

Note that there is one less metric for Condition dispatched tasks, because their dispatch latency can’t be measured.

2.7 Procedure Summary

section	Step	Action	Example / Location
2.1.1	Specify Task Parameters	Name of task to run	scanSensorReadings()
2.1.2		Milliseconds between executions	200
2.1.3		Short form name (4-8 char)	sensor
2.1.4		Very short label (1-4 char)	snsr
2.2	Generate the Variables	Macro loopTaskVars()	global_variables.cpp
2.3	Insert Initialization Code	setupTasks()	deviceSupport.cpp
2.4	Reset Code in Task	Reset timer or condition	Start of called task
2.5	Set Up oneSec Processing	Add 3 macro calls	Main.cpp, after loop()

2.8 Alerts Based on CPU Usage

The once per second processing checks the total CPU usage (first number displayed in its report line) an issues an Error trace if it is high. This trace may go to MQTT, or the console, or both, depending on the trace configuration.

If the CPU usage is over 90%, the following trace will be generated:

```
TraceEf("CPU Utilization RED alert = usage% =",totCPU);
```

If the CPU usage is between 60% and 90%, the following trace will be generated:

```
TraceEf("CPU Utilization YELLOW alert = usage% =",totCPU);
```

The global variable CPULoad can be used by functions which are able to defer processing to see if load shedding is needed. It has the following values:

```
int CPULoad;              // current state of CPU utilization from oneSec monitoring
const int CPUNormal = 0   // normal - below 60%
const int CPUYellow = 1   // yellow alert - 60-90%
const int CPURed = 2      // red alert, >90%
```

As well, a function is called when either of these alerts is issued so steps can be taken proactively to reduce CPU load:

```
void CPUOffLoad()         // attempt to reduce load on CPU
```

Currently, no action is taken by this function.