A SIMPLE TASK SCHEDULER FOR MICROCONTROLLERS

IN C++

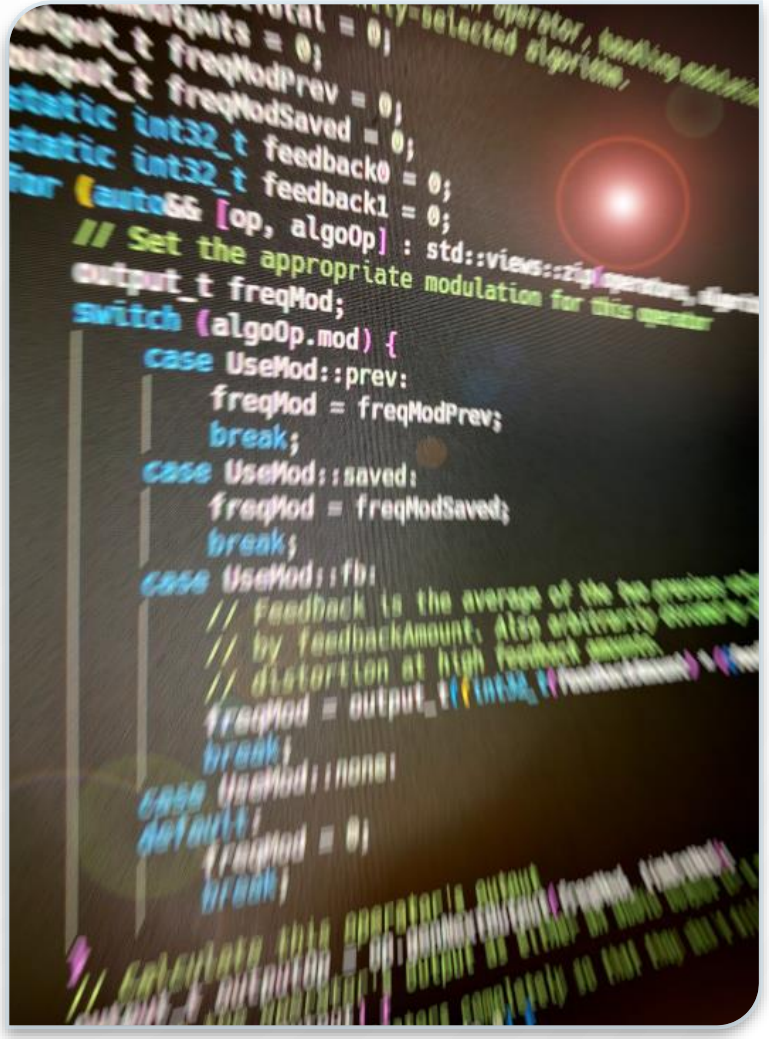
ABOUT ME

Len Popp

Retired Software Guy,

Software & Hardware Hobbyist

https://lenp.net/



AGENDA

- What It Is
- How to Use It
- How It Works
- Questions?

WARNING / PROMISE

Contains “modern” C++

```
[&](){}();
```

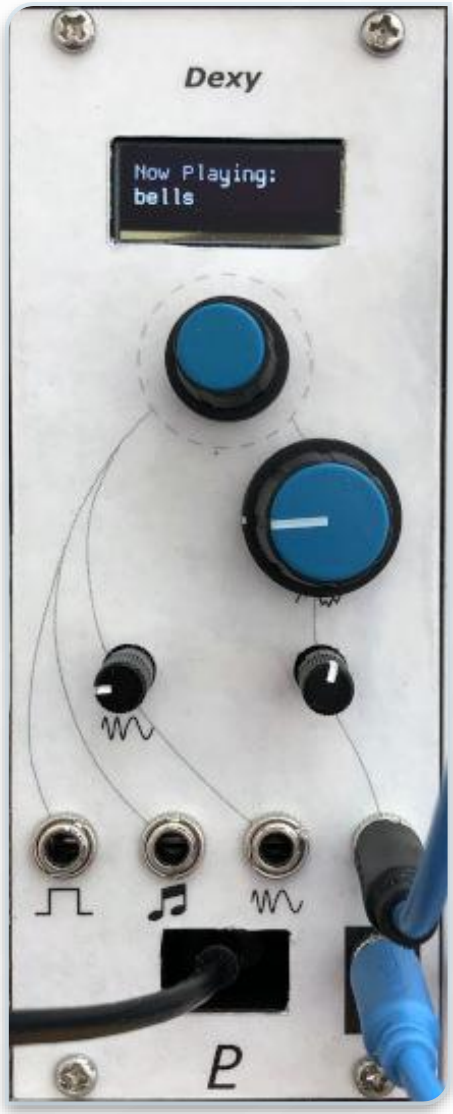
WHAT IT IS

BACKGROUND

- Embedded microcontroller
- Firmware written in C++
- Firmware must perform multiple tasks
- No operating system (RTOS)
  - Small-to-medium microcontroller
  - Programming with an SDK over the bare metal
- Want a simple task scheduler to help with simple projects

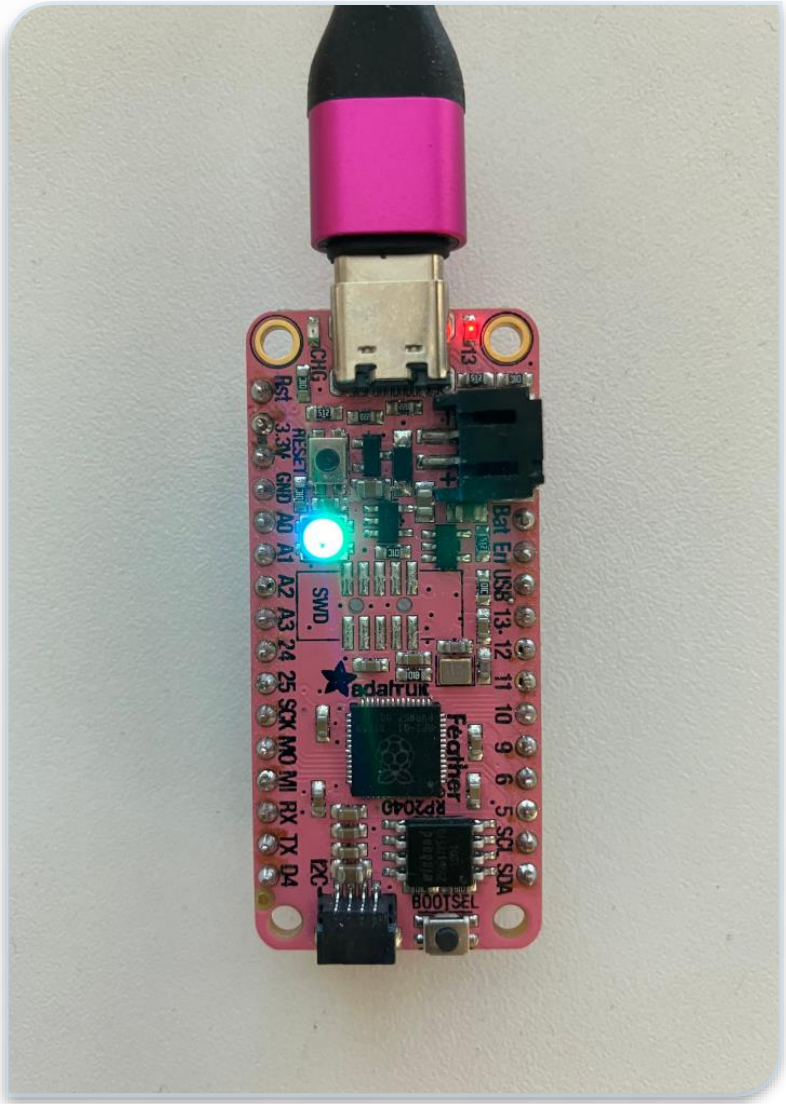
EXAMPLE: DEXY

- Digital FM synth module
- Based on a Raspberry Pi Pico(-ish)
- Firmware runs a task scheduler for low-priority tasks, including:
  - UI (display and rotary encoder)
  - USB serial I/O
  - Other tasks for testing, debugging, etc.
- Task scheduler is *not* used for audio synthesis
  - Interrupt-driven for timing precision
- https://lenp.net/synth/dexy/



DEMO

- Adafruit Feather RP2040
  - Raspberry Pi Pico-compatible board
- Two tasks:
  - Blinking LED
  - Colour-cycling RGB LED
- https://github.com/Len42/TaskDemo-RP2040



SIMPLE TASK SCHEDULING

- Several independent tasks to run “simultaneously”
- Round-robin execution
- Each task specifies how often it wants to run
- When called, each task performs an increment of work
- Very little overhead
  - Microcontroller’s performance and memory are limited

WHAT IT DOESN'T DO

- Not suitable for tasks with hard real-time requirements
- No task pre-emption
- No long-executing functions allowed
  - Tasks must be broken up into bite-size increments
  - For example, must use non-blocking calls for internet requests.
- Tasks not guaranteed to run precisely on time

HARDWARE REQUIREMENTS

- Not much!
- Clock/timer for elapsed time
  - Millisecond or microsecond resolution
- Interrupts *not* required

SOFTWARE REQUIREMENTS

- C++ 20
  - Can be implemented in “traditional” C++ with a bit more boilerplate code
- Elapsed time function
  - From device SDK

HOW TO USE IT

WRITING FIRMWARE WITH TASKS

MAIN()

- `Tasks::TaskList` defines the list of `Task` classes to be executed
  - Commented-out tasks are not compiled
- `initAll` calls each `Task`'s `init` function
- `runAll` calls each `Task`'s execute function
  - Called in order listed
- Loop forever

```
// Task List
constexpr Tasks::TaskList<
    //DebugTask,
    LedBlinkTask,
    LedColourTask
> taskList;


int main()
{
    // Initialize all the Tasks and run them forever
    taskList.initAll();
    while (true) {
        taskList.runAll();
    }
}
```

EXAMPLE TASK

- Task to blink an LED
- Class encapsulates everything, so this task is independent of other code
- Class is a subclass of `Tasks::Task`
- Implements 3 virtual functions
- Not necessary to declare an instance of `LedBlinkTask`
  - Done automatically by `Tasks::TaskList`

```
// LedBlinkTask: Task to blink an LED on & off
class LedBlinkTask : public Tasks::Task
{
public:
    // Task execution interval in microseconds
    unsigned intervalMicros() const override { return 500'000; }

    // Task initialization, called once at program start
    void init() override {
        gpio_init(PICO_DEFAULT_LED_PIN);
        gpio_set_dir(PICO_DEFAULT_LED_PIN, GPIO_OUT);
        gpio_put(PICO_DEFAULT_LED_PIN, false);
    }

    // Main task function
    void execute() override {
        fLed = !fLed;
        gpio_put(PICO_DEFAULT_LED_PIN, fLed);
    }

private:
    bool fLed = false;
};
```

HOW IT WORKS

IT'S NOT ROCKET SCIENCE

IT'S C++ MAGIC

TASK BASE CLASS

- Abstract base class
  - Pure virtual functions implemented by task subclasses
- `tick` is called frequently by the task scheduler
  - If it's time to execute this task, call `execute`
- `timer` keeps track of the next time to run the task
- `taskInstance` is a *variable template* that creates a single instance of each Task subclass

```
class Task
{
public:
    virtual unsigned intervalMicros() const = 0;

    virtual void init() = 0;

    virtual void execute() = 0;

    void tick(absolute_time_t now)
    {
        if (timeIsReached(now, timer)) {
            timer = make_timeout_time_us(intervalMicros());
            execute();
        }
    }

private:
    absolute_time_t timer = from_us_since_boot_constexpr(0);
};

template<typename TASK_T>
static TASK_T taskInstance;
```

TASK SCHEDULER

- This is the type of `taskList` seen earlier
- Template arguments are the Task subclasses
- Contains an array of `Task*`
- Constructor initializes the list of `tasks`
  - `consteval` – executes at compile time
- C++ magic turns a list of class names into a list of `Task` instances
- Iteration by fold expression over parameter pack

```
template<typename... TASKS>
class TaskList
{
public:
    consteval TaskList()
    {
        int i = 0;
        ((tasks[i++] = &taskInstance<TASKS>), ...);
    }

    void initAll() const /* ... */

    void runAll() const /* ... */

private:
    Task* tasks[sizeof...(TASKS)];
};

// in main.cpp (as seen before):
constexpr Tasks::TaskList<
    //DebugTask,
    LedBlinkTask,
    LedColourTask
> taskList;
```

TASK SCHEDULER (CONT.)

- Implements the `initAll` and `runAll` functions

```
template<typename... TASKS>
class TaskList
{
public:
    consteval TaskList() /* ... */

    void initAll() const
    {
        for (auto&& task : tasks) {
            task->init();
        }
    };

    void runAll() const
    {
        absolute_time_t now = get_absolute_time();
        for (auto&& task : tasks) {
            task->tick(now);
        }
    };

private:
    Task* tasks[sizeof...(TASKS)];
};
```

OUTRO

WHAT I LIKE

- Tasks are modular, independent
  - Good code organization
  - Easy to add and remove tasks (debugging etc.)
- Each task has its own repetition interval
- Low overhead
  - No heap memory allocation
  - A few bytes per task compared to inline code

WHAT I DON'T LIKE

- Not suitable for tasks with hard real-time constraints
  - A long-running task blocks other tasks from running

DOWNLOAD

https://github.com/Len42/TaskDemo-RP2040

FIN

QUESTIONS?