athreads - A Preemptive MCU Scheduler

Introduction

athreads is a small preemptive scheduling library for 8-bit AVR microcontrollers, developed and demonstrated on the ATmega2560.

The project implements a lightweight threading runtime in C and AVR assembly. It supports timer-driven preemption, context switching, per-thread time quanta, sleeping threads, runtime stack allocation from a stack pool, and basic CPU usage accounting.

The original idea was to understand how operating systems schedule tasks, but in a constrained embedded environment where there is no operating system underneath. Instead of only simulating scheduling on a PC, this project runs directly on a microcontroller and exposes scheduler behavior through hardware UI and serial profiling.

The project is useful as an educational low-level systems project because it shows how a preemptive runtime can be built from interrupts, stacks, registers, timers, and explicit context switching.

Project repository

General Description

The project is centered around the **athreads** scheduler library. The scheduler runs on the ATmega2560 and uses a hardware timer interrupt to periodically preempt the currently running thread.

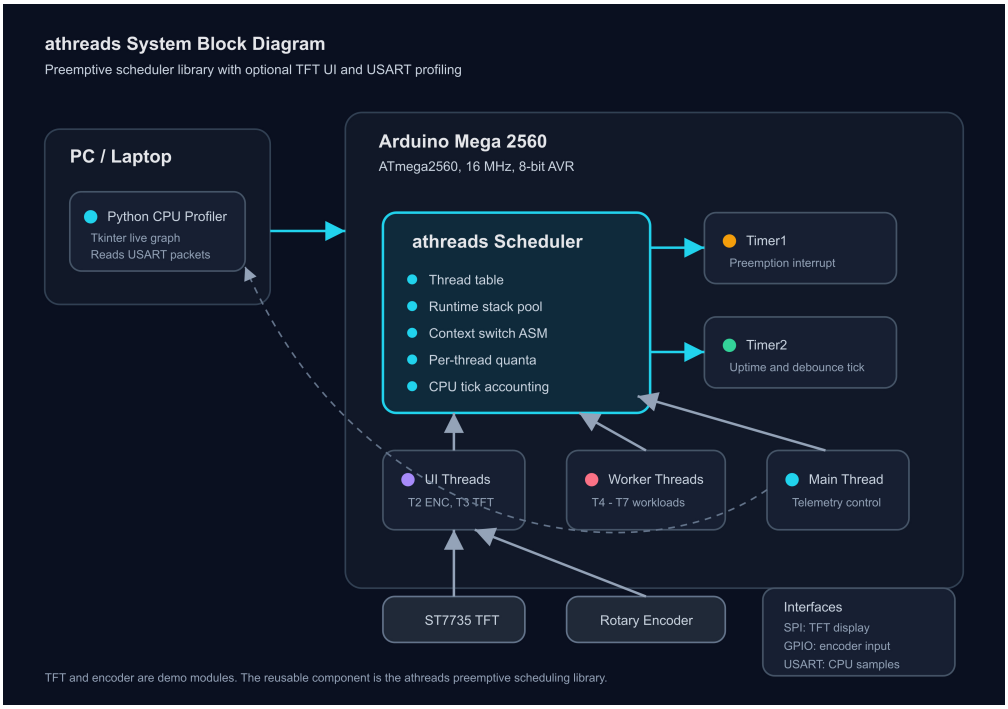
Each thread has:

- a thread descriptor;
- a saved stack pointer;
- a runtime-allocated stack region from the scheduler stack pool;
- a state;
- a time quantum;
- CPU tick accounting information.

The demo application adds:

- an SPI TFT color display for showing running threads;
- a rotary encoder for selecting a thread and changing its quantum;
- USART telemetry for sending CPU usage information to a PC;
- a Python Task Manager-style profiler that displays live per-thread CPU usage.

Block Diagram

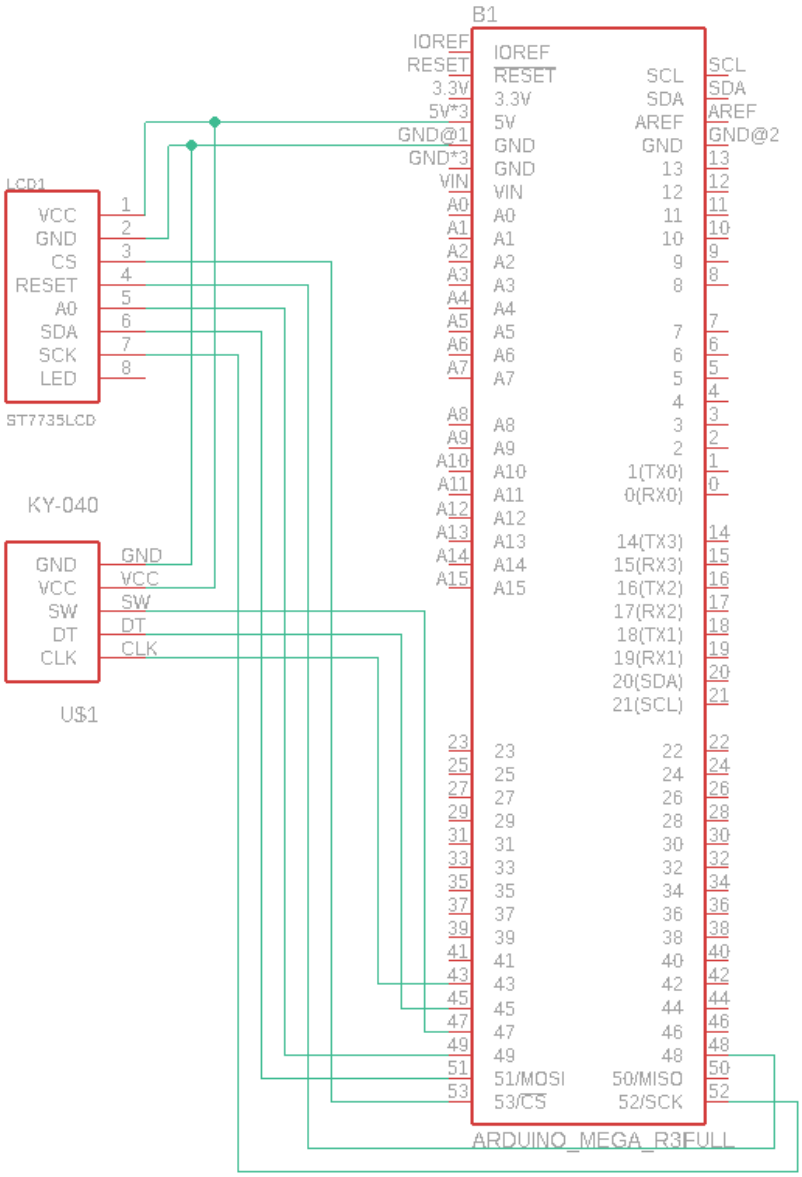


Hardware Design

Components

The hardware used for the demo consists of:

- Arduino Mega 2560 board with ATmega2560 microcontroller;
- 1.8 inch SPI TFT color display with ST7735 controller;
- rotary encoder module with push button;
- jumper wires and breadboard;
- USB cable for programming and serial communication.



Component Documentation / Datasheets

Component	Documentation / Datasheet
Arduino Mega 2560 Rev3	Arduino Mega 2560 documentation
Arduino Mega 2560 Rev3 board datasheet	Arduino Mega 2560 Rev3 datasheet PDF
Arduino Mega 2560 schematic	Arduino Mega 2560 schematic PDF
ATmega2560 microcontroller	Microchip ATmega2560 documentation and datasheet
ST7735R TFT controller	Sitronix ST7735R datasheet
KY-040 rotary encoder module	KY-040 rotary encoder datasheet PDF

TFT Display Wiring

The current display is a 1.8 inch SPI TFT color display using an ST7735 controller.

TFT Pin	Arduino Mega 2560 Pin
SCL / SCK / CLK	D52 / SCK
SDA / MOSI / DIN	D51 / MOSI

RST / RES	D48
DC / A0	D49
CS	D53
VCC	5V
GND	GND
BL / LED	5V

The module used in this project is compatible with 5V systems, so it can be connected directly to the Arduino Mega 2560 pins.

The ST7735 module used during testing required display inversion mode. The firmware enables `INVON`, and the driver uses pre-inverted RGB565 color constants so that the physical display appears in dark mode correctly.

Rotary Encoder Wiring

Encoder Pin	Arduino Mega 2560 Pin
SW	D47
DT	D45
CLK	D43
+	5V
GND	GND

Hardware Role

The TFT display and rotary encoder are not required by the scheduler library itself. They are part of the demonstration layer.

The TFT shows the currently running threads and their quantum values. The encoder allows the user to navigate through the thread list, enter edit mode, and modify the selected thread's quantum while the scheduler is running.

Software Design

Development Environment

The project is developed using:

- C for most firmware code;
- AVR assembly for context switching;
- PlatformIO for building and uploading firmware;

- Python for the optional desktop CPU profiler;
- Tkinter for the profiler GUI.

Project Structure

```
include/
  athread/      Public scheduler API and generated AVR structure offsets
  platform/     USART, uptime, and debug headers
  profiling/    CPU statistics, tracing, and worker demo headers
  ui/           TFT display and encoder headers

src/
  athread/      Scheduler implementation and AVR context switch assembly
  platform/     USART and millisecond uptime support
  profiling/    CPU sampling, trace hooks, and demo workloads
  ui/           ST7735 SPI TFT driver and rotary encoder input
  main.c        Demo firmware entry point

tools/
  cpu_task_manager.py       Live CPU profiling viewer
  gen_offsets.py            PlatformIO pre-build offset generator
  gen_athread_offsets.c     Offset generator source
```

Scheduler Design

The scheduler keeps a table of thread descriptors. Each descriptor stores:

- thread ID;
- thread state;
- entry function;
- saved stack pointer;
- stack boundaries;
- sleep counter;
- scheduling metadata.

Threads are created with an explicit stack size. The scheduler allocates stack memory from a static stack pool at runtime.

This avoids hardcoding one global array per thread, such as:

```
static uint8_t main_stack[MAIN_STACK_SIZE];
static uint8_t worker_stack[WORK_STACK_SIZE];
```

Instead, the scheduler owns a single pool:

```
static uint8_t stack_pool[ATHREAD_STACK_POOL_SIZE];
```

```
static uint16_t stack_pool_used;
```

When a thread is created, the scheduler reserves a slice of this pool and stores the stack boundaries in the thread descriptor.

Preemption

Timer1 is used for preemption. When the Timer1 compare interrupt fires, the scheduler updates the running thread's quantum counter. If the quantum expires, the current thread context is saved and another ready thread is selected.

Timer2 is used for millisecond uptime and periodic support work, including sleeping-thread wakeups and encoder debouncing.

Public Scheduler API

Function	Description
`athread_init()`	Initializes scheduler state, creates the idle thread, and resets the stack pool.
`athread_start()`	Starts the scheduler. After this call, execution is controlled by the scheduler and the function does not normally return.
`athread_create(entry, info, stack_size)`	Creates a new thread, allocates `stack_size` bytes from the scheduler stack pool, initializes its context, and returns the thread ID or `ATHREAD_INVALID_TID` on failure.
`athread_yield()`	Voluntarily gives up the CPU and allows another ready thread to run.
`athread_sleep_ticks(ticks)`	Puts the current thread to sleep for a number of scheduler ticks.
`athread_set_quantum(tid, quantum_ticks)`	Changes the time quantum of a thread at runtime.
`athread_get_quantum(tid)`	Returns the configured time quantum of a thread.
`athread_get_thread_count()`	Returns the number of allocated thread IDs, including the idle thread.
`athread_get_cpu_ticks(out_ticks, max_ticks)`	Copies per-thread CPU tick counters for profiling and diagnostics.
`athread_get_current_tid()`	Returns the ID of the currently running thread.
`athread_tick()`	Advances scheduler timing state, including sleeping-thread wakeups. Used by the platform timer code.
`athread_bootstrap()`	Internal startup wrapper used when a thread begins execution.

Example thread creation:

```
uint8_t tid = athread_create(worker_thread, worker_info, 512);
```

Quantum

A quantum is the amount of scheduler time a thread is allowed to run before it can be preempted.

In this project, the quantum is measured in scheduler timer ticks. For example, if the scheduler tick is 5 ms, a quantum of 4 allows a thread to run for approximately 20 ms before the scheduler may switch to another thread.

Changing a thread's quantum affects responsiveness and CPU distribution:

- a larger quantum gives a thread longer uninterrupted execution;
- a smaller quantum makes scheduling more responsive, but increases context switching overhead.

TFT User Interface

The TFT display shows a dark-mode process menu. It displays thread IDs, names, quantum values, and small quantum bars.

The rotary encoder is used as follows:

- rotate to move through the thread list;
- press to enter or leave quantum edit mode;
- rotate in edit mode to increase or decrease the selected thread's quantum.

The display driver uses ST7735 SPI commands and RGB565 color values. Because this module required inversion mode, the driver enables `INVON` and uses pre-inverted colors internally.

Profiling

The profiling system is optional and is built on top of the scheduler's CPU counters.

The firmware periodically reads per-thread CPU tick counters and sends compact packets over USART. On the PC, a Python program reads the serial stream and displays a live graph of CPU usage per thread.

The viewer can be started with:

```
python ./tools/cpu_task_manager.py --port COM3
```

If the default Python installation does not include Tkinter, a Python installation with Tkinter support must be used.

Demo Threads

The demo firmware creates several threads to show scheduling behavior:

Thread	Name	Purpose
T0	IDLE	Idle thread created by the scheduler
T1	MAIN	Main application coordinator
T2	ENC	Rotary encoder input handling
T3	TFT	TFT process menu
T4	WRK1	Synthetic worker workload
T5	WRK2	Synthetic worker workload
T6	WRK3	Synthetic worker workload
T7	WRK4	More dynamic worker workload

Results

The project successfully implements a working preemptive scheduler on the ATmega2560.

The main achieved results are:

- multiple independent threads can run on an 8-bit AVR microcontroller;
- threads are preempted using a hardware timer interrupt;
- thread contexts are saved and restored using AVR assembly;
- stacks are allocated from a scheduler-managed stack pool;
- thread stack sizes are selected when creating each thread;
- threads can voluntarily yield or sleep;
- time quanta can be changed at runtime;
- per-thread CPU usage can be measured;
- scheduler behavior can be visualized live on a PC;
- thread state can be inspected and modified using the TFT and encoder UI.

The profiling viewer makes it possible to observe how scheduling decisions affect CPU distribution between threads.

Conclusions

This project helped me understand preemptive scheduling from the hardware level upward. Implementing context switching on a microcontroller made the relationship between stacks, registers, interrupts, timers, and scheduling much clearer than a high-level simulation would have.

The most challenging parts were the AVR assembly context switch, stack initialization for newly created threads, runtime stack pool management, and making profiling work without disturbing the system too much.

The final result is a small but functional preemptive scheduling library, with optional profiling and

hardware visualization tools that make the runtime behavior visible and easier to reason about.

Possible future improvements include:

- configurable scheduling policies;
- priority-based scheduling;
- mutexes and synchronization primitives;
- better stack usage diagnostics;
- stack overflow detection;
- portability to other AVR boards or other MCU families.

Download

The project source code, firmware, tools, and documentation are available in the repository:

athreads project repository

Build command:

```
pio run
```

Upload command:

```
pio run -t upload
```

Run the CPU profiler:

```
python ./tools/cpu_task_manager.py --port COM3
```

Bibliography / Resources

Hardware Resources

- Arduino Mega 2560 Rev3 documentation
- Arduino Mega 2560 Rev3 datasheet
- Arduino Mega 2560 schematic
- Microchip ATmega2560 documentation and complete datasheet
- Sitronix ST7735R TFT controller datasheet
- KY-040 rotary encoder module datasheet

Software Resources

- PlatformIO documentation
- avr-gcc documentation
- Python Tkinter documentation
- Microchip AVR documentation portal

Export to PDF