Investigation Report on Multi-MCU Multi-Process

中文 | English

Team Members

- Weikang Wei
- Jiahong Luo
- Zhuo Cui
- Yanzhen Guo

1. Research Background

中文 | English

Embedded systems are widely used in various devices, most of which have only one processor core and typically run a single task. To run multiple tasks, the most primitive method is to use interrupts to interrupt the main thread's execution, thereby handling sudden events. Based on timer interrupts, there are a series of embedded OSs that can achieve pseudo (through scheduling) multitasking. We plan to design a computational system with multiple processors, where each processor can run multiple tasks, as well as the required OS software, to meet the needs of robots and other devices.

1. Bare-metal Programming (No Operating System)

- **Single-task Sequential Execution**:
  In simple systems without an operating system, the MCU executes code sequentially through a `while(1)` loop, essentially running a single task.
- **Pseudo-multitasking**:
  Simulating multitasking through interrupts and state machines. For example:
  i. **Interrupt Service Routine (ISR)**: Handles high-priority tasks (such as button detection, communication reception).
  ii. **State Machine in Main Loop**: Time-slices processing of multiple tasks (such as display refresh, sensor data acquisition). **Still single-threaded**, but logically divided into multiple

tasks.

2. Using Real-time Operating Systems (RTOS)

- **Multitasking Concurrency**:
  RTOSs (such as FreeRTOS, uC/OS, ThreadX) allow multiple tasks (threads) to be scheduled concurrently on a single-core MCU through **time slicing** or **priority preemption**.

3. Typical Scenario Examples

- **Without RTOS**: Logically divided into multiple functional modules but physically runs as a single thread.
- **With RTOS**: STM32F4 + FreeRTOS can stably run 5~20 tasks (depending on task resource requirements).
- **High-performance MCU**: STM32H7 dual-core can handle two high-load tasks in parallel, supplemented by multiple low-priority tasks.

4. Related Project Research

FreeRTOS

FreeRTOS is a very popular open-source real-time operating system widely used in various embedded systems, including STM32 microcontrollers.

The project can directly use its scheduler and communication mechanisms. However, it lacks distributed support and requires manual expansion.

Features:

- Supports task management, semaphores, queues, and more.
- Optimized for resource-constrained devices.
- Provides extensive documentation and community support.
- STM32 Support: FreeRTOS provides support for STM32 and integrates FreeRTOS configuration options in the STM32CubeMX tool, facilitating rapid development.

Zephyr Project

Zephyr is an extensible small real-time operating system suitable for resource-constrained devices and IoT applications.

Features:

- Supports multiple architectures, including ARM Cortex-M.
- Built-in support for multiple wireless technologies such as Bluetooth and WiFi.
- STM32 Support: Zephyr supports STM32 series microcontrollers, making it suitable for applications requiring complex network functions.

ThreadX

ThreadX is also a real-time operating system.

Features:

- Extremely low interrupt latency and fast task switching.
- Includes middleware components such as file systems and USB support.
- Provides detailed documentation and technical support.

CMSIS-RTOS

CMSIS-RTOS V2 is a standard API provided by ARM to simplify the porting of RTOS across different embedded systems.

Features:

- Standardized APIs make code easier to port.
- Supports the latest ARM architecture.
- Seamlessly integrates with Keil MDK.
- STM32 Support: CMSIS-RTOS V2 is designed for ARM architecture, thus naturally supporting STM32 series microcontrollers.

StarPU

StarPU is a unified runtime system for heterogeneous multicore architectures, see Zhuo Cui's OSH Research Report

2. Step-by-step Project Implementation Plan

Phase 1: Setting Up Basic Toolchain and Test Platform

Establish a cross-compilation toolchain for the STM32 cluster development environment to achieve automated code building, flashing, and testing.

1. Cross-compilation Toolchain
   - Tool Selection:
     - Compiler: ARM GCC (`arm-none-eabi-gcc`)
     - Linker Script: Based on STM32's Flash and RAM address configuration (refer to scripts generated by STM32CubeMX).
     - Flashing Tool: OpenOCD + ST-Link/V2 debugger.
   - Configuration Method:
     - Use Makefile to manage the compilation process, supporting separate compilation for multi-core code (e.g., generating independent binary files for different processors).
2. Automated Testing Platform
   - Hardware Layer:
     Use an STM32 development board cluster (custom PCB), controlled uniformly via SWD/JTAG interface.
   - Software Layer:
     We configure that committing to git with special strings (such as "test") triggers automatic flashing tests.

Phase 2: Single Core Single Task

Traditional MCU mode testing is sufficient.

Phase 3: Dual-Core Dual-Task Communication

In this phase, we need to handle shared memory issues between two cores.

Phase 4: Single Core Multi-Task Scheduling

1. Scheduler Design:

- Task Control Block (TCB): Stores task status, stack pointer, priority, etc.
- Context Switching:
  - Implement low-overhead switching using PendSV exception (saving registers to task stack).
  - Assembly code example (ARM Cortex-M):

```
PendSV_Handler:
    CPSID I                 ; Disable interrupts
    MRS R0, PSP             ; Get current task stack pointer
    STMDB R0!, {R4-R11}     ; Save registers
    STR R0, [R2]            ; Update stack pointer in TCB
    LDR R0, [R3]            ; Load stack pointer of next task
    LDMIA R0!, {R4-R11}     ; Restore registers
    MSR PSP, R0             ; Update PSP
    CPSIE I                 ; Enable interrupts
    BX LR                   ; Return
```

2. Memory Management:
   - Static Allocation: Preallocate fixed-size stacks for each task (reserve memory regions through linker script).
   - Dynamic Allocation: Implement simple memory pool.

Phase 5: Multi-Core Extension and Overall Testing

1. Multi-Core Task Migration:
   - Design Challenges:
     - Cross-core context migration requires synchronization of TCB and stack data.
     - Interrupts need to be redirected to the target core.
   - Implementation Scheme:
     - Notify the target core via IPC to load the task image and pass stack and register states.
2. Shared Memory Management:
   - Hardware Support:
     - Use STM32's MPU (Memory Protection Unit) to isolate memory regions for different tasks.
   - Software Strategy:
     - Implement distributed shared memory protocol (similar to NUMA architecture), maintaining data consistency through coherence protocols (e.g., MESI).
3. Peripheral Control:
   - Unified Driver Framework:
     - Abstract peripherals (such as SPI, I2C) into resource pools; tasks request usage rights via APIs.

3. Feasibility Analysis

中文 | English

1. Core Architecture

- Microkernel Design:
  - Implement a lightweight kernel providing essential services (scheduling, IPC, memory management) to minimize overhead.
  - User-space services (e.g. drivers, protocols) run as independent threads for modularity.

2. Multi-Threaded Interaction

- Thread Management:
  - Preemptive Scheduler: Priority-based round-robin scheduling to balance task priorities and ensure fairness.
  - Context Switching: Optimize efficiency using Thread Control Blocks (TCBs) storing register states and stack pointers.
  - Thread Synchronization: Local mutexes/semaphores for shared memory; atomic operations for low-level locks.

3. Memory Sharing

- Local Memory:
  - Heap Management: Use buddy systems or slab allocators for dynamic allocation.
  - Shared Regions: Define memory zones with access control via kernel-managed mutexes.
- Distributed Memory:
  - Global Address Space: Map remote memory into a virtual address range (e.g. `0x8000_0000+` for remote MCUs).
  - Proxy-Based Access: Transparent read/write via kernel messages (e.g. SPI/CAN packets with MCU ID, address, data).
  - Caching: Optional LRU caching for frequent remote accesses; explicit invalidation ensures coherence.

4. Inter-MCU Communication

- Protocol Layer:
  - Transport: Use CAN bus or SPI; packets include source/destination IDs, read/write commands, address, and data.
  - Message Queues: Prioritize messages (e.g., control vs. data) and handle inbound packets via interrupts.
- APIs:
  - `remote_read(mcu_id, address)`
  - `remote_write(mcu_id, address, data)`

5. Distributed Scheduling

- Decentralized Coordination:
  - Task Pools: Global task queues accessible via atomic operations; MCUs fetch tasks when idle.
  - Work Stealing: MCUs steal tasks from neighboring queues to balance load.
- Fault Tolerance:
  - Heartbeat Monitoring: Detect offline MCUs and redistribute their tasks.
  - Checkpointing: Periodically save task states to non-volatile memory for recovery.

6. Cross-MCU Synchronization

- Distributed Lock Manager (DLM):
  - Mutexes assigned a "home" MCU to manage lock requests.
  - Messages for `lock()`, `unlock()`, and grant/deny responses.
  - Timeout mechanisms to prevent deadlocks.

7. Developer APIs

- Threads:
  - `thread_create(entry_func, priority)`
  - `thread_yield()`, `thread_sleep(ms)`
- Memory:
  - `shm_alloc(size)`, `shm_free(addr)`
  - `remote_map(mcu_id, remote_addr, local_addr)`

- Synchronization:
  - `mutex_init()`, `mutex_lock()`, `mutex_unlock()`
  - `barrier_wait()`, `message_send(mcu_id, data)`

8. Security & Reliability

- Access Control: Per-MCU permissions for remote memory regions.
- CRC/Checksums: Ensure data integrity in communication.
- Watchdog Timers: Reboot unresponsive MCUs.

9. Workflow

1. Thread Creation:
   - Thread A on MCU1 writes to local shared memory.
   - Thread B on MCU2 requests a remote lock via DLM.
2. Remote Access:
   - MCU2's kernel sends a CAN message to MCU1 to lock the mutex.
   - Upon approval, Thread B reads/writes via `remote_read()` / `remote_write()`.
3. Task Migration:
   - MCU3's scheduler steals a task from MCU2's queue via work stealing.

10. Optimization

- Resource Limits: Enforce thread count and heap size restrictions per MCU.
- Testing: Use simulated MCU networks to debug race conditions.

Tools & Prototyping

- Base RTOS: Extend FreeRTOS/Zephyr with custom IPC and scheduling logic.
- Hardware: Use ARM Cortex-M or ESP32 with CAN/SPI support.

This design balances scalability, real-time performance, and resource constraints, enabling collaborative multi-microcontroller systems.

4. References

- Bare Metal STM32 Programming Part 11: Using External Memories
- Communication between Multiple Microcontrollers - Electrical Engineering Stack Exchange
- STM32 FreeRTOS Task Scheduling Tutorial with Example Code - Embedded There
- Distributed Operating System - Wikipedia
- Microcontroller Based Distributed and Networked Control System for Public Cluster - ResearchGate
- A Survey of Distributed Real-Time Operating Systems for Embedded Systems
- Microcontroller-Based Distributed Control Systems: Challenges and Solutions
- Efficient Task Scheduling for Multi-Microcontroller Systems in IoT Applications
- Design and Implementation of a Distributed Embedded System Using STM32 Microcontrollers
- Cluster-Based Multi-Core Scheduling for Real-Time Embedded Systems
- A Lightweight Distributed Operating System for IoT Edge Devices
- Real-Time Multi-Microcontroller Communication Using CAN Bus in Embedded Systems
- Embedded Systems for Robotics: Multi-Core Approaches to Motion Control
- Optimizing Distributed Embedded Systems with Shared Memory and Multi-Core Architectures
- Edge Computing with Microcontroller Clusters: A Case Study on IoT Applications