关于多单片机多进程的调研报告

中文 | English

项目成员

- 位文康
- 罗嘉宏
- 崔卓
- 郭彦禛

1. 研究背景

中文 | English

嵌入式系统广泛使用于各类设备中，而它们通常只有一个处理器核心，也一般只运行一个任务。为了运行多个任务，最原始的方式是使用中断来打断主线程的运行，实现对突发事件的处理。在定时器中断的基础上，有一系列的嵌入式 OS，可以实现伪 (通过调度的) 多任务。而我们计划为了满足机器人等等设备的需求，设计一个包含多处理器且每个处理器上可以运行多任务的计算系统，以及所需的 OS 软件。

1. 裸机编程（无操作系统）

- **单任务顺序执行**：
  在无操作系统的简单系统中，MCU 通过一个 `while(1)` 依次按顺序执行代码，**本质上是单任务**。
- **伪多任务**：
  通过中断和状态机模拟多任务。例如：
  i. **中断服务程序（ISR）**：处理实时性高的任务（如按键检测、通信接收）
  ii. **主循环中的状态机**：分时处理多个任务（如显示刷新、传感器采集）。**实际仍为单线程**，但可逻辑上划分为多个任务。

2. 使用实时操作系统（RTOS）

- **多任务并发**：
  RTOS（如FreeRTOS、uC/OS、ThreadX）允许在单核 MCU 上通过**时间片轮转**或**优先级抢占**调度多个任务（线程）。

3. 典型场景示例

- **无RTOS**：逻辑上划分多个功能模块，但物理上单线程运行。
- **RTOS**：STM32F4 + FreeRTOS 可稳定运行 5~20 个任务（取决于任务资源需求）。
- **高性能MCU**：STM32H7双核可并行处理两个高负载任务，辅以多个低优先级任务。

4. 相关项目调研

FreeRTOS

FreeRTOS 是一个非常流行的开源实时操作系统，广泛应用于各种嵌入式系统中，包括STM32微控制器。

项目可直接使用其调度器和通信机制。但缺乏分布式支持，需自行扩展。

特点:

- 支持任务管理、信号量、队列等功能。
- 对资源受限设备进行了优化。
- 提供了丰富的文档和支持社区。
- STM32 支持: FreeRTOS 提供了对 STM32 的支持，并且在 STM32CubeMX 工具中集成了 FreeRTOS 配置选项，便于快速开发。

Zephyr Project

Zephyr 是一个可扩展的小型实时操作系统，适用于资源受限设备和物联网应用。

特点:

- 支持多种架构，包括 ARM Cortex-M。
- 内置对蓝牙、WiFi等多种无线技术的支持。
- STM32 支持: Zephyr 对 STM32 系列微控制器提供了支持，适合需要复杂网络功能的应用场景。

ThreadX

ThreadX 也是一个RTOS。

特点:

- 极低的中断延迟和快速的任务切换。
- 包含文件系统、USB支持等多个中间件组件。
- 提供详细的文档和技术支持。

CMSIS-RTOS

CMSIS-RTOS V2 是一个由 ARM 提供的标准 API，旨在简化 RTOS 在不同嵌入式系统的移植。

特点:

- 标准化的 API 使得代码更易于移植。
- 支持最新的 ARM 架构。
- 可以与 Keil MDK 无缝集成。
- STM32 支持: CMSIS-RTOS V2 专为 ARM 架构设计，因此自然支持STM32系列微控制器。

StarPU

StarPU 是一个异构多核架构的统一运行时系统，详见崔卓的OSH调研报告

2. 项目分步实现方案

阶段 1：搭建基础工具链与测试平台

为STM32集群开发环境建立交叉编译工具链，实现代码自动化构建、烧录和测试。

1. 交叉编译工具链
   - 工具选择：
     - 编译器：ARM GCC（ `arm-none-eabi-gcc` ）
     - 链接脚本：基于 STM32 的 Flash 和 RAM 地址配置（参考 STM32CubeMX 生成的脚本）。
     - 烧录工具：OpenOCD + ST-Link/V2 调试器。

- 配置方法：
  - 使用 Makefile 管理编译流程，支持多核代码分离编译（例如为不同处理器生成独立的二进制文件）。

2. 自动化测试平台
   - 硬件层：
     使用 STM32 开发板集群（定制 PCB ），通过 SWD/JTAG 接口统一控制。
   - 软件层：
     我们设置在 git commit 时加入特殊字符串（如 test ）会触发自动烧录测试。

阶段 2：单核单任务

传统 MCU 模式测试即可。

阶段 3：双核双任务通信

本阶段我们需要处理两核间共享内存的问题。

阶段 4：单核多任务调度

1. 调度器设计：
   - 任务控制块（TCB）：存储任务状态、堆栈指针、优先级等信息。
   - 上下文切换：
     - 使用 PendSV 异常实现低开销切换（保存寄存器到任务堆栈）。
     - 汇编代码示例（ ARM Cortex-M ）：

```
PendSV_Handler:
    CPSID I                 ; 关闭中断
    MRS R0, PSP             ; 获取当前任务堆栈指针
    STMDB R0!, {R4-R11}     ; 保存寄存器
    STR R0, [R2]            ; 更新TCB中的堆栈指针
    LDR R0, [R3]            ; 加载下一个任务的堆栈指针
    LDMIA R0!, {R4-R11}     ; 恢复寄存器
    MSR PSP, R0             ; 更新PSP
    CPSIE I                 ; 开启中断
    BX LR                   ; 返回
```

2. 内存管理：
   - 静态分配：为每个任务预分配固定大小的堆栈（通过链接脚本保留内存区域）。

- 动态分配：实现简易内存池。

阶段 5：多核扩展与整体测试

1. 多核任务迁移：
   - 设计挑战：
     - 跨核上下文迁移需同步 TCB 和堆栈数据。
     - 中断需重定向到目标核。
   - 实现方案：
     - 通过 IPC 通知目标核加载任务镜像，并传递堆栈和寄存器状态。
2. 共享内存管理：
   - 硬件支持：
     - 使用 STM32 的 MPU（Memory Protection Unit）隔离不同任务的内存区域。
   - 软件策略：
     - 实现分布式共享内存协议（类似 NUMA 架构），通过一致性协议（如 MESI ）维护数据同步。
3. 外设控制：
   - 统一驱动框架：
     - 将外设（如 SPI、I2C ）抽象为资源池，任务通过 API 申请使用权。

3. 可行性分析

中文 | English

1. 项目核心架构

- 微内核设计：
  - 实现轻量级内核，提供基础服务（调度、进程间通信、内存管理），以最小化开销。
  - 用户空间服务（如驱动、协议）作为独立线程运行，以实现模块化。

2. 多线程交互

- 线程管理：
  - 抢占式调度器：基于优先级的轮询调度，平衡任务优先级，确保公平性。
  - 上下文切换：通过线程控制块（ TCB ）存储寄存器状态和栈指针，优化切换效率。
  - 线程同步：本地互斥锁/信号量用于共享内存；原子操作用于低级锁。

3. 内存共享

- 本地内存：
  - 堆管理：使用伙伴系统或 slab 分配器进行动态分配。
  - 共享区域：通过内核管理的互斥锁划分受控访问的内存区域。
- 分布式内存：
  - 全局地址空间：将远程内存映射到虚拟地址范围（例如 `0x8000_0000+` 表示远程 MCU ）。
  - 代理访问：通过内核消息（如包含 MCU ID、地址和数据的 SPI/CAN 数据包）实现透明读写。
  - 缓存：可选 LRU 缓存优化高频远程访问，显式失效机制确保一致性。

4. 微控制器间通信

- 协议层：
  - 传输：使用 CAN 总线或 SPI 协议；数据包包含源/目标 ID、读写命令、地址和数据。
  - 消息队列：按优先级（如控制 vs 数据）分类消息，通过中断处理入站数据包。
- API：
  - `remote_read(mcu_id, address)`
  - `remote_write(mcu_id, address, data)`

5. 分布式调度

- 去中心化协调：
  - 任务池：全局任务队列通过原子操作访问；MCU 空闲时可获取任务。
  - 工作窃取：MCU 从邻居队列“窃取”任务以平衡负载。
- 容错机制：
  - 心跳监测：检测离线 MCU 并重新分配其任务。
  - 检查点：定期将任务状态保存到非易失性存储，支持故障恢复。

6. 跨 MCU 同步

- 分布式锁管理器（ DLM ）：
  - 互斥锁分配“归属” MCU 管理锁请求。
  - 消息支持 `lock()` 、 `unlock()` 以及授权/拒绝响应。
  - 超时机制防止死锁。

7. 开发者 API

- 线程：
  - `thread_create(entry_func, priority)`
  - `thread_yield()` 、 `thread_sleep(ms)`
- 内存：
  - `shm_alloc(size)` 、 `shm_free(addr)`
  - `remote_map(mcu_id, remote_addr, local_addr)`
- 同步：
  - `mutex_init()` 、 `mutex_lock()` 、 `mutex_unlock()`
  - `barrier_wait()` 、 `message_send(mcu_id, data)`

8. 安全与可靠性

- 访问控制：按 MCU 分配远程内存区域的权限。
- CRC / 校验和：确保通信数据完整性。
- Watchdog Timers：重启无响应的 MCU 。

9. 工作流

1. 线程创建：
   - MCU1 的线程 A 写入本地共享内存。
   - MCU2 的线程 B 通过 DLM 请求远程锁。
2. 远程访问：
   - MCU2 内核向 MCU1 发送 CAN 消息以锁定互斥锁。
   - 接受授权后，线程 B 通过 `remote_read()` / `remote_write()` 读写。
3. 任务迁移：
   - MCU3 的调度器通过“工作窃取”从 MCU2 队列获取任务。

10. 优化

- 资源限制：限制每个 MCU 的线程数量和堆大小。
- 测试：使用模拟的 MCU 网络调试竞态条件。

工具与原型设计

- 基础 RTOS：基于 FreeRTOS 或 Zephyr 扩展自定义 IPC 和调度逻辑。
- 硬件：使用支持 CAN/SPI 的 ARM Cortex-M 或 ESP32 。

此设计在可扩展性、实时性能和资源约束之间取得平衡，支持多微控制器协作系统。

4. 引文

- 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