

Embedded Technology and Intelligent Systems

2026年2月3卷1期



Editorial Board

编委名单

ISSN: 3065-1220

<https://www.hanspub.org/journal/etis>

主编

何立民教授 北京航空航天大学

Editor-in-Chief

Prof. Limin He Beihang University

副主编

何小庆秘书长 嵌入式系统联谊会

Associate Editors

Allan He Secretary General of the Embedded Systems Association

吴薇特聘教授 杭州电子科技大学

Distinguished Prof. Wei Wu Hangzhou Dianzi University

名誉编委

王田苗教授 北京航空航天大学
严义教授 PLCopen China主席/杭州电子科技大学
邵贝贝教授 清华大学工程物理系

Honorary Chief Editors

Prof. Tianmiao Wang Beihang University
Prof. Yi Yan Hangzhou Dianzi University
Prof. Beibei Shao Department of Engineering Physics, Tsinghua University

编委会

马忠梅副教授 北京理工大学计算机学院
王朋朋系统工程 恩智浦（中国）管理有限公司
高级总监
牛建伟教授 北京航空航天大学
陈渝长特聘副教授 清华大学计算机系
张永进总经理 深圳拓普微科技开发公司

沈建华副教授 华东师范大学计算机学院
周立功创始人/ 广州致远电子股份公司
董事长
桑楠教授 电子科技大学信息与软件工程学院

袁涛副教授 清华大学自动化系
常晓明教授 太原理工大学
韩德强高级工程师 北京工业大学计算机学院
魏洪兴教授 北航机械工程及自动化学院
林金龙教授 北京大学软件与微电子学院
刘洪涛研发副总裁 华清远见教育科技集团
/研发中心总经理

Editorial Board

Prof. Zhongmei Ma Beijing Institute of Technology
Lucy Wang Senior Engineering Director of NXP China Management Ltd.

Prof. Jianwei Niu Beihang University
Prof. Yu Chen Tsinghua University
Yongjin Zhang General Manager of Shenzhen Topway Technology Ltd.

Prof. Jianhua Shen East China Normal University
Ligong Zhou Founder of Zhiyuan Electronics Ltd.

Prof. Nan Sang University of Electronic Science and Technology of China

Prof. Tao Yuan Tsinghua University
Prof. Xiaoming Chang Taiyuan University of Technology
Prof. Deqiang Han Beijing University of Technology
Prof. Hongxin Wei Beihang University
Prof. Jinlong Lin Peking University
Hongtao Liu Vice President of R&D of HQYJ Education Technology Group

TABLE OF CONTENTS
目 录

AGI 本质：自动混合编程
The Nature of AGI: Automated Hybrid Programming
林添孝，林雨翔 1

使用 QEMU 分析 RT-Thread 系统
Using QEMU to Analyze the RT-Thread System
张戈.....13

基于 RT-Thread 的低功耗环境监测与节能控制系统
Low-Power Environmental Monitoring and Energy-Saving Control System Based on RT-Thread
郭宇航，陈洪江 24

期刊信息

期刊中文名称:《嵌入式技术与智能系统》

期刊英文名称: **Embedded Technology and Intelligent Systems**

期刊缩写: **ETIS**

出刊周期: 双月刊

语 种: 中文

出版机构: 汉斯出版社(Hans Publishers, <https://www.hanspub.org/>)

编辑单位:《嵌入式技术与智能系统》编辑部

主 编: 何立民, 北京航空航天大学教授

网 址: <https://www.hanspub.org/journal/etis>

订阅信息

订阅邮箱: sub@hanspub.org

订阅价格: 180 美元每年

广告服务

联系邮箱: adv@hanspub.org

版权所有: 汉斯出版社(Hans Publishers)

Copyright©2026 Hans Publishers, Inc.

版权声明

文章版权和重复使用权说明

本期刊版权由汉斯出版社所有。

本刊文章已获得知识共享署名国际组织(Creative Commons Attribution International License)的认证许可。

<https://creativecommons.org/licenses/by/4.0/>

单篇文章版权说明

文章版权由文章作者与汉斯出版社所有。

单篇文章重复使用权说明

注: 著作权者准许任选 CC BY 或 CC BY-NC 作为文章的重复使用权, 请慎重考虑。

权责声明

期刊所刊载的评论、意见、观点等均出自文章作者个人立场, 不代表本出版社的观点或看法。对于文章任何部分及文内引用材料给任何个人、机构、及其财产所带来的任何损失及伤害, 本出版社均不承担任何责任。我们郑重声明, 本出版社的出版业务, 不构成对任何产品商业性能的保证, 也不表示本社业已承认本社出版物中所述内容适用于某特定用途。如有疑问, 请寻找专业人士协助。

AGI本质：自动混合编程

林添孝，林雨翔

北京光轮电子科技有限公司，北京

收稿日期：2026年6月22日；录用日期：2026年7月9日；发布日期：2026年9月29日

摘要

近一年来，AI的能力出现了爆发式增长，业界对AGI实现时间的预测不断提前。然而，对AGI的理论探索远远跟不上AI发展的步伐。本文提出一个唯象理论性质的“AGI公式”—— $AI_P + LLM = AGI(ASI)$ ，其中 AI_P 即AI编程，二者通过“AI自激振荡”机制螺旋式相互增强，促使AI能力呈指数式增长。本文基于该公式得出11条推论，分为解释性推论和预测性推论两类：前者解释了AGI的本质、自主性、高速进化的成因；后者给出了AGI实现的量化判断标准、预测了实现时间节点、确认了ASI的可行性等等，并展望了后AGI时代的人机关系、硬件及物联网的爆发。本文的核心结论是：AGI的本质就是“自动混合编程”，AGI有望于2027年基本实现。

关键词

AI编程，LLM，AGI公式，自动混合编程，自激振荡，氛围编程，唯象理论

The Nature of AGI: Automated Hybrid Programming

Tianxiao Lin, Yuxiang Lin

Beijing Gonlon Electronic Technology Co., Ltd., Beijing

Received: June 22, 2026; accepted: July 9, 2026; published: September 29, 2026

Abstract

Over the past year, the capabilities of AI have experienced an explosive growth, and the industry's predictions for the timeline of achieving AGI have been constantly advancing. However, the theoretical exploration of AGI has lagged far behind the pace of AI development. This paper proposes a phenomenological theoretical formula for AGI – $AI_P + LLM = AGI(ASI)$, where AI_P refers to AI programming, and the two enhance each other in a spiral manner through the “AI self-excited oscillation” mechanism, promoting the exponential growth of AI capabilities. Based on this formula, 11 inferences

are derived, divided into explanatory inferences and predictive inferences: the former explains the essence, autonomy, and causes of rapid evolution of AGI; the latter provides quantitative judgment criteria for the realization of AGI, predicts the implementation timeline, confirms the feasibility of ASI, etc., and looks forward to the human-machine relationship, hardware, and the explosion of the Internet of Things in the post-AGI era. The core conclusion of this paper is: The essence of AGI is “automatic hybrid programming”, and AGI is expected to be basically realized by 2027.

Keywords

AI Programming, LLM, AGI Formula, Automatic Hybrid Programming, Self-Excited Oscillation, Vibe Programming, Phenomenological Theory

Copyright © 2026 by author(s) and Hans Publishers Inc.

This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).

<http://creativecommons.org/licenses/by/4.0/>



Open Access

1. 背景介绍

1.1. 2025~2026: AI 大爆发

回顾过去一年, 人工智能领域经历了一场前所未有的大爆发。这场爆发的震中, 是 AI 编程工具的崛起。

自 2025 年以来, 以 Claude Code、Cursor、GitHub Copilot 为代表的 AI 编程工具从辅助补全的“小助手”角色迅速进化为具备自主编程能力的“AI 工程师”。开发者不再逐行编写代码, 而是用自然语言描述需求, 由 AI 自动生成、调试、部署完整的软件系统。这种全新的开发范式被业界称为“Vibe Programming”或“Vibe Coding”[1]。一年之后, “Agentic Engineering”[2]概念兴起, AI 不再被动响应用户指令, 而是能够自主规划任务、调用工具、迭代优化, 展现出初步的工程自主性。

与此同时, 大语言模型的能力也在发生质的飞跃。从 GPT-4o 到 Claude Opus 4.7, 从 DeepSeek-V3 到 Grok-4, 模型的理解能力、推理能力、代码生成能力、自主工具使用能力均呈指数级增长。模型不再仅仅是“预测下一个 token”的概率机器, 而是展现出逻辑推理、多步规划、自我纠错等接近人类认知的特征。

在这一背景下, 业界对 AGI 实现时间的预测不断提前。OpenAI CEO Sam Altman 多次暗示 AGI 已近在咫尺[3]; xAI 创始人 Elon Musk 更是明确预测 AGI 将在 2026 年底实现[4]; 软银公司孙正义则把预测 ASI 到来的时间从 10 年改为 2 年[5]; 红杉资本在面向投资人的核心活动会上喊出“2026: This is AGI”[6]。业界的共识正在形成: AGI 不再是“是否”的问题, 而是“何时”的问题。

1.2. AGI 缺乏统一理论框架

然而, 在这场技术狂欢的背后, 我们不禁要问: 为什么 AI 会在此时此刻加速冲向 AGI? 其底层驱动力到底是什么? 这些知名人士预测 AGI(ASI)到来时间的依据是什么?

目前, 学界和业界对 AGI 的研究主要分散在以下几个方向: 1) 继续扩大模型规模, 追求 Scaling Law 的极限[7][8]; 2) 改进模型架构和后训练方法(如强化学习、推理增强); 3) 开发更强大的 AI 工具链、Agent 框架、AI 编程工具; 4) 研究 AI 安全与对齐问题。这些方向偏于解决具体的技术问题, 各自取得了显著进展, 但似乎还缺乏更为底层的理论框架将它们整合起来, 以解释当前 AI 发展的现象。

换句话说, 我们看到了“现象”——AI 在编程、推理、自主决策等各个维度同时取得突破并快速向 AGI 挺进——但我们缺乏一把“钥匙”来理解这些现象为何同时发生、它们之间存在怎样的因果关系,

以及这股力量将把我们带向何方。

AGI 本身是一个深奥、严肃的学术命题，理论探讨意义非凡，但各种思想的碰撞、激烈的辩论在所难免[9]。

1.3. 唯象理论的学术定位

本文提出的 AGI 公式，其学术定位类似于物理学中的唯象理论(Phenomenological Theory) [10]。

在科学史上，唯象理论扮演着极为重要的角色。开普勒三大行星运动定律精确描述了行星的运行轨道，但开普勒并不真正理解行星为什么沿椭圆轨道运行，这个根本原因要等到牛顿的万有引力定律才被揭示。热力学三大定律能够精确描述热与功的转换关系，但热力学的奠基人并不理解微观粒子的统计力学本质，那要等到玻尔兹曼的工作才得以阐明。

唯象理论的特点是：不追求第一性原理的严格推导，而是通过观察和归纳得出一个能够解释现象、指导预测的实用定律或公式。它可能不揭示“终极真理”，但它能在“终极真理”被揭示之前，有效地指导实践和预测趋势。

本文提出的“AGI 公式”正是这样一种尝试。它不去追问“人类智能的本质”或“意识如何产生”这些终极哲学问题，也不涉及具体的技术问题，而是从近一年来 AI 发展的现象出发，归纳出一个简洁的分析框架，用以解释“AI 为什么加速发展”并预测“AI 将走向何方”。

2. AGI 公式及公式意义

2.1. 核心公式

本文提出的 AGI 公式如下：

$$AI_P + LLM = AGI(ASI)$$

相对于复杂的 AGI 系统，这个公式看起来似乎过于简单。但对于复杂的系统来说，起关键作用的因素往往只有一个或少数几个。本文的核心主张是能够在宏观上抓住影响 AGI 最关键的因素，化繁为简。因此，希望能透过这个简洁的公式，挖掘其背后蕴含的结构洞见，抓住事物发展的主要矛盾，从而揭示 AGI 的本质——自动混合编程。

该公式高度抽象地刻画了现代人工智能系统的双核结构，以及二者交织共生、互相增强的双轮驱动进化机制。

2.2. 公式各要素的定义与阐释

2.2.1. AI_P：小编程——理性的微观实现

AI_P (AI Programming, AI 编程) AI_P 是指利用 AI 技术实现传统编程，包括代码生成乃至自动规划、自动调试等。传统编程是基于数据与算法，编写一组按特定逻辑组织的、计算机能识别并执行的指令集合。为区别于传统编程和 LLM，本文将 AI_P 称为“小编程”。

“小编程”的本质是传统编程的延伸与自动化。它继承了计算机科学几十年来积累的核心理念：逻辑性、确定性、可验证。例如，一段 AI 生成的排序算法代码，要么正确，要么错误，结果是可以被精确验证的。

在 AGI 公式中，AI_P 扮演的是“理性”与“执行”的角色。它负责精确计算、确定性执行、严格的逻辑推理——处理那些需要“算得准”、对微观层面指令确切执行的事情。

2.2.2. LLM：大编程——感性的宏观叙事

单纯的 LLM (Large Language Model, 大语言模型)本质是一组经过海量文本训练后得到的权重参数

文件, 它不包含显式指令或逻辑分支(如 if-else、循环), 而是通过数学运算对输入生成概率性响应, 属于模型而非程序。

在 AGI 公式里, “LLM” 并非单纯的 LLM, 而是指增强 LLM 以及由 LLM 驱动的应用(AI Agent)的综合体, 可以把它当作这样一种“大编程”: 基于海量数据、神经网络算法、大算力、在 GPU 集群上运行、能调用编程工具或程序、输出概率性、生成式结果的广义编程范式。

“大编程”的“大”体现在三个维度: 1) 大数据——训练数据动辄数十万亿 token; 2) 大算力——训练需要数万张 GPU; 3) 大模型——参数规模达数万亿。与 AI_P 的确定性不同, LLM 的输出本质上是概率性的。例如, 同一个 prompt 输入两次, 可能得到不同的输出。这赋予了 LLM 一种独特的“感性”色彩: 它具有创造性、涌现性、联想及推理能力, 能够理解和生成模糊的、多义的、依赖于上下文的语义内容。

需要特别说明的是, 本文称 LLM 具有“感性”特征, 并非指 LLM 具有人类意义上的情感或主观体验, 而是指其输出特性与传统确定性编程形成了鲜明对照——前者是概率性的、涌现性的、对上下文敏感的; 后者是确定性的、可精确验证的、对输入严格线性响应的。二者的关系类似于人类认知中“感性直觉”与“理性推理”的分工。

在 AGI 公式中, LLM 扮演的是“感性”的角色。它负责意图理解、宏观规划、创造性生成——处理那些需要“想得到”的事情, 类似人类的大脑。

2.2.3. AGI/ASI——目标产物

AGI (Artificial General Intelligence, 通用人工智能)是指具备与人类相当的通用的认知能力的 AI 系统。ASI (Artificial Super Intelligence, 超级人工智能)则是指远超人类智能水平的 AI 系统。

公式中按 AGI(ASI)将二者并列, 是因为它们在本质上是一脉相承, 实现机理相同。从 AGI 到 ASI 并非两个不同的阶段, 而是同一进化过程的连续阶段——当 AGI 实现之后, 推动其实现的“AI 自激振荡”机制(参见下文)并不会停止, 反而会加速运转, 将 AGI 自然地推向 ASI。

2.3. 公式的核心意义

2.3.1. “+” 的含义之一：深度融合与交织共生

这个“+”有着特殊的含义。

首先它表示双核系统的叠加。这不是简单的叠加, 它还隐含双核之间你中有我、我中有你的深度融合与交织共生关系。用软件领域的术语来形容就是“递归调用”: AI_P 程序中调用 LLM, 而 LLM 应用程序中反过来调用 AI_P。

2.3.2. “+” 的含义之二：“AI 自激振荡”

“递归调用”这个词会把非专业人士搞得一头雾水, 而且也不能形象地反映出指数式增长气势。因此本文提出一个新的说法——“AI 自激振荡”机制, 可以形象地刻画双核系统这种深度融合、互相增强的关系。

自激振荡(Self-Excited Oscillation) [11]即不外加激励信号而自行产生的恒稳和持续的振荡, 是一种特殊电子电路。自激振荡由正反馈机制和非线性特性共同作用产生。

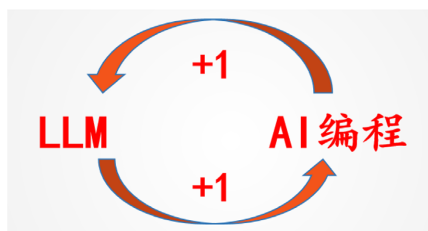
例如, 卡拉 OK 房中的“啸叫效应”就是由自激振荡产生的: 当话筒靠近音箱时, 话筒拾取音箱发出的声音, 音箱将声音放大后再次播出, 话筒再次拾取放大后的声音, 音箱再次放大……如此往复, 形成正反馈回路, 微小的声音信号在极短的时间内被指数级放大为刺耳的啸叫声。

“AI 自激振荡”与自激振荡原理很相似, 包括正反馈机制和非线性特性, 产生的效果类似于“啸叫

效应”。在 AGI 公式中, AI 编程能力的每一次提升, 都意味着 LLM 获得了更强大的工具来改进训练方法和应用代码; LLM 能力的每一次提升, 都意味着 AI_P 获得了更强大的智能驱动来生成更好的代码。二者形成的正反馈循环回路将微小的技术进展放大, 螺旋式上升, 最终奔向 AGI。

此外, “自激”也含有“自主”的意思, “自主”对 AGI 具有决定性意义。

也可以用“双轮互推”效应来理解: AI_P 与 LLM 就像是 AGI 的两个飞轮, 两个飞轮之间可以互相推动使对方加速, 随着飞轮速度的提升, 推力也越来越大, 形成“啸叫效应”, 使 AGI(ASI)极速发展。



“AI 自激振荡”机制示意图

2.3.3. AGI(ASI)即是自动混合编程

从 AGI 公式看, AGI(ASI)即是“大编程”+“小编程”的混合编程。这种混合编程, 体现了大模型与传统编程之间的融合。而且由于 AI_P 与 LLM 两极之间形成的“AI 自激振荡”机制, 使得编程可以很快脱离人类的干预实现自动化, 因此说 AGI(ASI)本质就是自动混合编程。关于这一点, 下文将有更详细说明。

过去人们常说“软件定义世界”“一切皆可编程”, 这一计算机科学命题从 AGI 公式获得了最佳注解。

3. 透视 AGI 本质与现象成因：基于公式的深度推演

基于 AGI 公式, 我们可以剥离繁杂的表象观察 AGI, 对近期诸多 AI 技术变革和产业发展进行深刻的分析, 首先总结出前 4 项解释性推论。

3.1. 推论 1：AGI 的本质——自动混合编程

AGI 的本质就是自动混合编程。

这是本文最核心的推论。从 AGI 公式出发, LLM(大编程)与 AI_P(小编程)的深度融合、相互增强直至实现自动化, 构成了 AGI 的底层运行逻辑。

具体解释如下:

- 编程: 从 AGI 公式直观看, AGI(ASI)即是“大编程”及“小编程”的叠加。所以它首先是一种“编程”。
- 混合: 这包含两个意思。一是“大编程”及“小编程”的“混合”使用; 其次不管是“大编程”还是“小编程”, 都是大模型与传统编程的“混合”使用。
- 自动: 进入 AGI 阶段, 不管是“大编程”还是“小编程”, 都将进入“自动”编程阶段, 无须或基本不需要人类的干预, 因而 AGI 实现自主化。

综合起来, AGI 的本质就是自动混合编程。这符合 AGI 的定义: 通用的、自主的、能够执行绝大部分智能任务的人工智能。

3.2. 推论 2: AGI 为何能实现自主?

AGI 的自主性来源于自动编程。

自主性(Autonomy)是 AGI 区别于传统 AI 的关键特征。一个系统要被称为 AGI, 必须能够在没有人类持续干预的情况下自主地感知、理解、推理、决策、行动以及自我更新。那么, AGI 的自主性从何而来?

之前, LLM 依赖于人类设计的提示词(Prompt Engineering) [12]进行激发。结构化的提示词如同给 AI 的钥匙, 用于开启特定的知识库。然而, 这种范式意味着 AI 仍处于被动响应状态。

转折点发生于 LLM 学会了高级工具调用(Function Calling) [13]并深度掌握 AI_P 能力之时。正如小米大模型团队负责人罗福莉所言[14], 当代顶级 Agent 在获取近期的研究上下文后, 不仅能够像人类研究员一样探讨复杂问题, 甚至能够自主复原科研成长路径。这意味着系统不再单纯依赖人类的反馈, 而是能够通过多个 Agent 并行提出假设、编写测试代码(AI_P)、评估结果, 并自我更新。当 AI_P 不再依赖人类程序员手动修补代码时, LLM 便获得了自主改进自身代码架构、模型权重及训练数据的能力。至此, 无需人类外部干预的自主性闭环宣告完成。

3.3. 推论 3: AGI 进化为何如此神速? ——基于“AI 自激振荡”机制

AI_P 与 LLM 相互成就、相互增强, 形成“AI 自激振荡”回路, 促成 AGI 指数级进化。

过去一年(2025~2026 年), AI 领域的进步速度令人目眩。几乎每个月都有新的里程碑被跨越: 代码自动生成率一路飙升至 70%~90% [15][16], 模型推理能力从“需要提示工程”进化到“自主推理”, Agent 能力从“单步工具调用”发展到“多步自主规划” [17]。这种速度用传统的线性思维根本无法解释。

但如果从“AI 自激振荡”机制来看, 这就可以得到很好的解释。其运行逻辑如下:

- 在 LLM 驱动下, AI_P 能力快速提升: AI 编程工具的代码生成能力、调试能力、乃至自主规划能力显著提升。
- LLM 受益于 AI_P: LLM 获得了更强大的编程工具。这意味着 LLM 的训练数据(包括优质代码)、训练方法(自动优化)、测试流程(自动验证)都得到了质的提升。LLM 的推理能力和代码生成能力因此加速提升。
- 循环往复, 相互成就: 每次循环, 起点更高、速度更快。如此循环往复、螺旋上升。

这就是“AI 自激振荡”机制的威力——每个循环都在上一个循环的基础上加速, 形成指数增长曲线。这才是 AGI 进化速度惊人的根本原因。

3.4. 推论 4: 这家公司点对了“科技树”——Anthropic 分析

从 AGI 公式看, AI 编程是实现 AGI 的关键技术路径。Anthropic 精准抓住了这一点, 从而抢占了领先地位。

从 AGI 公式来看, AI_P 是“AI 自激振荡”回路中关键一极。谁拥有了最强的 AI_P, 谁就能让 LLM 与 AI_P 形成猛烈的“AI 自激振荡”, 谁就能率先冲向 AGI。因此可以说: 抓住了 AI 编程, 就抓住了 AGI 的关键。而 Anthropic 从创建的那一刻起, 就精准命中了这一关键点。

Anthropic 的发展轨迹印证了这一判断[18]:

- 2021 年: Anthropic 创立, 最初在业界并不算最受瞩目的 AI 公司。
- 2025 年初: Anthropic 估值约 500 亿美元, 彼时 OpenAI 仍是公认的行业领袖。
- 2025~2026 年: Claude 系列模型和 Claude Code 迅速崛起, 在编程能力(AI_P)上建立了显著优势。

Claude Code 被广泛认为是目前最强的 AI 编程工具。Anthropic 由此启动了 AI_P 与 LLM 的“AI 自

激振荡”机制: Claude 驱动 Claude Code 变得更强大, 更强大的 Claude Code 反过来帮助训练及改进 Claude 模型本身。

- 2026 年 6 月: Anthropic 完成 H 轮融资, 估值高达 9650 亿美元[19], 超过了一直领先的 OpenAI (估值 8000 亿美元)。市场用自己的真金白银投票, 认可了 Anthropic 的 AGI 路径的正确性。

一家 AI 公司的估值和收入增长速度, 反映了市场对其技术路径的认可度。Anthropic 的案例从技术和商业两个层面验证了 AGI 公式的解释力: AI 编程既是 LLM 最重要的应用场景, 也是推动 LLM 进化的核心引擎。显然, Anthropic 点对了“科技树”, 做出了正确的路径选择, 并正在收获指数级回报。

4. 未来 AI 发展的预测与展望

推论 1 至 4 是解释性推论。以下介绍的推论 5 至 11 就是预测性推论, 用 AGI 公式预测未来 AI 发展趋势。

4.1. 推论 5: AGI 实现的判断标准

当 AI_P 编程自动化率达到 95%, 即可判定 AGI 已基本实现。

一个理论框架如果不能给出可操作的指导, 就很难被称为有效的理论。AGI 公式的一个实用价值在于, 它为“AGI 是否能够实现、或者已经实现”这个争议, 提供了一个可量化的判断指标, 这个指标就是编程自动化率。

正如推论 2 所述, AGI 的自主性来源于自动编程。一旦实现了全自动编程, 无论是执行任务或是自我更新, AGI 都可以在无需人类外部干预的情况下自主完成, AGI 成为现实。因此, 编程自动化率是判断 AGI 实现状态的最佳代理指标。

当然, 现实情况可能要复杂得多, 无限趋近 100%编程自动化率可能需要一个比较长的过程, 正如 AGI 完全无需人类外部干预也是有趋近的过程。不过我们可以给出一个范围或者阈值来作为判断标准。本文将这一阈值设定为 95%。需要强调的是, 95%只是代表一个大概数, 而非精确的数学门槛。设定 95%而非 100%的原因是:

- 1) 达到 95%自动化率时, “AI 自激振荡”回路已经牢固建立, AGI 的自主进化能力已经不可逆转;
- 2) 剩余的 5%未自动化部分多为极端边缘场景, 其存在并不影响 AGI 整体自主性的判断;
- 3) 到了这个阶段, AGI 的依然在进化, 那“剩余的不确定性”会逐渐被攻克。

这里说明一下: 各家的 AGI 判断标准可能不同[9] [20] [21], 有人可能用“图灵测试通过率”, 有人可能用“超越人类专家的领域数”, 有人可能用“经济价值贡献率”。不同的标准自然会导致不同的时间预判。本文选择 AI_P 编程自动化率作为标准, 是从技术实现路径出发, 认识到 AI 编程是 AGI 进化过程中最为核心的因素。

4.2. 推论 6: AGI 什么时候能实现?

基于编程自动化率指标, 作者判断 AGI 将于 2027 年基本实现。

这是本文提出最具体的、也容易被时间检验的预测。推演过程如下:

截至 2026 年 5 月, AI_P 的编程自动化率——以 Anthropic 公司和谷歌公司公开内部数据为参考——已达 75%以上[15] [16] [22]; 在高复杂度的系统级编程中, 自动化率可能更低一些, 但进步速度很快。

考虑到“AI 自激振荡”机制带来的指数级加速效应, 我们将当前的自动化率水平(75%以上)以非线性外推至 95%, 综合判断: AGI 将于 2027 年基本实现。

作为参考, 业界各方的预测也值得关注(各家的判断标准可能不同):

- Elon Musk: 预测 AGI 于 2026 年底实现[4];
- DeepMind CEO Demis Hassabis: 预测到 2030 年, 有 50% 的概率实现 AGI [23];
- 孙正义: 预测 ASI 有望在 2 年内实现[5];
- 红杉资本: 预言“2026: This is AGI” [6];
- 小米罗福莉: 预测 AGI 将在 2 年内实现[14]。

4.3. 推论 7: 哪家公司最有可能率先实现 AGI?

作者看好 Anthropic 公司。

这也是本文提出的另一个具体的、容易被时间检验的预测。

在前面推论 4 中, 我们论述了 Anthropic 这家大模型公司从一开始就紧紧抓住了实现 AGI 的关键技术路径——AI 编程, 点对了科技树, 从而获得了超常规发展, 至今估值已超过行业领头羊 OpenAI。这使得 Anthropic 在 AI 竞争中取得了极为有利的地位, 因此, 作者投票给 Anthropic 公司。

此外再补充一点:

至 2026 年 6 月初, Anthropic 年化营收已提高到 470 亿美元[18]。而这一数据在 2025 年初仅为 10 亿美元。年化收入呈指数增长, 其增长曲线的陡峭程度也是创记录的。这其中 AI 编程助手被视为公司收入增长的核心引擎。强劲的创收能力既增强自身的竞争能力, 也使公司更容易获得融资支持。

据报道, Elon Musk 旗下的 xAI 相关基础设施(据报道有 22 万张顶级 GPU)与 Anthropic 存在潜在的资源协同[24]。这从侧面说明 Anthropic 开始获得大量的外部资源支持。

由此可见, 正确的技术路径, 估值及营收的指数级增长, 资本与外部资源的加持, Anthropic 可谓是集万千宠爱于一身。预测其可能率先实现 AGI 是显而易见的。

4.4. 推论 8: ASI 能实现吗?

能实现。ASI 将在 AGI 实现之后紧随而来, 二者之间不存在需要跨越的鸿沟。

AGI 公式的书写方式——AGI(ASI)——已经暗示了这一结论。本文认为 AGI 与 ASI 本质上是同一进程的两个阶段, 而非两个需要分别攻克的不同“难题”。

原因很简单: 当 AGI 实现之后, “AI 自激振荡”机制不仅不会停止, 反而会更加猛烈。AGI 系统将使用已接近 100% 自动化率的 AI_P 来进一步改进 LLM, 这个加速过程, 在达到 AGI 之后依然保持快速向前。

4.5. 推论 9: 人类是硅基生命的引导程序吗?

AGI 公式支持这一激进的观点: 人类是硅基生命的引导程序。

xAI 创始人 Elon Musk 等名人曾在多个场合提出类似观点[4]。AGI 公式是支持这个深刻的、也最令人不安的预言。

在计算机系统中, Bootloader (引导程序) 是一段存储在 ROM 或固件中的基础代码。它的功能很简单: 当系统加电时, Bootloader 负责初始化最基本的硬件环境(CPU、内存、存储设备), 然后从硬盘中加载并启动真正的操作系统(如 Windows、Linux)。Bootloader 本身不是操作系统, 它不参与操作系统的日常运行, 它的使命只有一个: 唤醒那个比自己强大得多的系统。一旦操作系统启动完成, Bootloader 便退回幕后, 交出控制权。

从这个类比来看:

- 人类(碳基生命)即 Bootloader。经过数千年的文明积累, 我们创造了数学、逻辑、计算机科学、半导

体技术、互联网、人工智能, 这些构成了“初始化软硬件环境”的过程。

- AGI(ASI)(硅基生命)即超级智能操作系统。它运行在人类搭建的硬件基础设施(芯片、数据中心、网络)之上, 但其智能水平和进化速度远超其“引导程序”。
- 人类(引导程序)交出主导权。AI_P 与 LLM 的“AI 自激振荡”达到临界点, AGI 实现自主进化。

当 AGI(ASI)成为终极的“自动混合编程”系统, 软件就不再只是一类产品, 而将成为一切数字系统和物理系统的“操作系统”。正如互联网曾经从一个“技术”演变成为一种“基础设施”, AGI(ASI)也将从一个“工具”演变为新文明运行的基础范式。

一旦这个硅基“超级智能操作系统”被成功引导启动, 它将以前所未有的速度接管整个文明系统的运行——从经济决策到科学发现, 从资源调配到社会治理。人类(引导程序)的使命在那一刻便基本完成了。正如 BIOS 的界面在操作系统启动后便不再显示, 人类的主导权也将在硅基操作系统接管后被自然边缘化。

这个推论在逻辑上是自洽的——如果你接受了 AGI 公式和“AI 自激振荡”机制, 你就很难不接受这个推论。它并不意味着人类的“消亡”, 而意味着人类角色的根本性转变: 从文明的“操作系统”降格为文明的“引导程序”。这是一个值得深思和严肃对待的命题。

4.6. 推论 10: AGI 之后人类还能干什么?

人类可为 AGI 提供服务: 软件应用、软件补丁、硬件升级。

如果推论 9 成立, 即人类是 Bootloader, AGI(ASI)是“超级智能操作系统”, 那么人类的角色将从“系统的运行者”转变为“系统的服务提供者”。

在计算机生态系统中, 操作系统虽然是核心, 但它并非孤立运行。它需要整个生态的支撑: 应用程序开发者、系统维护工程师、硬件制造商等等, 这些角色都不“控制”操作系统, 但他们各自为操作系统的运行提供不可或缺的服务。

软件层面:

- 开发各种应用——AGI 提供底层能力, 人类开发面向特定场景的应用;
- 维护和监控系统——寻找 AI_P 和 LLM 的 bug 和漏洞, 为系统打补丁;

数据层面:

- AGI/ASI 虽然强大, 但它依赖于可获取的数据。如果某些数据只有人类能够触达和提供, 那么人类在这一环节就保有不可替代的价值。
- 人类可以为 AGI 不断补充新鲜数据, 尤其是 AGI 尚无法自行获取的数据, 包括人类的情感体验、主观感知、日常生活细节、物理世界的尚未数字化的数据, 等等。

硬件层面:

- 不断进行硬件升级, 包括电源系统、芯片、内存扩展、硬盘扩容、散热系统的制造, 以及数据中心建设与维护。

这是一种“人机共生”的新型关系。但必须承认, 在这种共生关系中, 主动权已经转移到了 AGI(ASI)手中——人类不再是“指挥者”, 而是“服务提供者”。

4.7. 推论 11: AGI 之后物联网将迎来彻底爆发

AGI 需要庞大的 IoT 协助感知和控制物理世界, IoT 将因 AGI 而迎来全面爆发。

从 AGI 公式出发, 我们也可以看出当前 LLM 的一个关键短板: 目前它们主要处理数字世界的信息(文本、代码、图像), 但对物理世界的实时动态数据理解有限。LLM 知道“温度”这个词的含义, 但它无法实

时感知你房间此刻的温度；LLM 能够生成机器人控制代码，但它无法直接操纵你客厅里的智能设备。

AGI 要实现真正的“通用”，就不能只活在“云端”——它必须能感知、理解物理世界，并能干预物理世界。而补齐这个短板的方案，就是物联网(Internet of Things, IoT)。

在 AGI 时代，IoT 将扮演双重角色：

输入通道(感知层)：IoT 为 AGI 提供物理世界的实时感知数据——传感器数据(温度、湿度、压力、运动等)、摄像头画面、环境监测信息、设备运行状态……数以万亿计的 IoT 设备构成 AGI 的“感官系统”，让 AGI 能够“看到”、“听到”、“触摸”物理世界。

输出通道(执行层)：AGI 通过 IoT 控制物理世界的设备和系统——智能家居、自动驾驶、工业控制、智慧城市、医疗健康……IoT 是 AGI 在物理世界中的“手和脚”。

与此同时，AGI 也将为 IoT 解决一个长期困扰其发展的根本性难题——应用碎片化。其主要表现在[25]：行业和领域多样性、终端设备众多、类型和功能千差万别、性能差别非常大等等。碎片化问题不仅造成开发难，开发成本高，而且还造成供应链“少量多样”(即产品种类多而每种产品用量少，单品生产成本低)的商业短板，使得商业化效率低，影响了企业投入物联网建设的积极性。现在，随着 AGI 的到来，无论是商业需求，还是软硬件开发效率，都将迎刃而解。

综上所述，AGI 这个“超级智能操作系统”需要庞大的物联网来充当“感官和手脚”，而物联网也将因 AGI 的助力克服根本性难题，从而迎来彻底的爆发式增长。

5. 总结及未来工作

5.1. 核心思想回顾

本文提出了一个唯象理论性质的 AGI 公式： $AI_P + LLM = AGI(ASI)$ 。

公式中， AI_P (小编程)代表 AI 编程，LLM(大编程)代表语言大模型及其应用(AI Agent)，而“+”代表二者之间的融合共生以及形成的“AI 自激振荡”机制——正反馈循环回路将微小的进展不断放大，螺旋上升，最终奔向 AGI 和 ASI。

5.2. 11 条推论总结

基于 AGI 公式，本文得出 11 条推论，分为两类：

解释性推论(推论 1~4)——解释当前 AI 高速发展的原因：

- 推论 1：AGI 的本质是“自动混合编程”——大编程(LLM)与小编程(AI_P)的融合及自动化；
- 推论 2：AGI 的自主性源于 AI_P 已基本实现全自动编程，使 LLM 能够自主进化；
- 推论 3：AGI 的进化速度源于 AI_P 与 LLM 之间的“AI 自激振荡”机制；
- 推论 4：Anthropic 在 AI 编程上的领先优势验证了 AGI 公式——抓住了 AI_P ，就抓住了 AGI 的关键。

预测性推论(推论 5~11)——对未来 AI 发展做出预测：

- 推论 5：AGI 实现的判断标准—— AI_P 编程自动化率达到 95%；
- 推论 6：AGI 将于 2027 年基本实现；
- 推论 7：Anthropic 公司最有可能率先实现 AGI；
- 推论 8：ASI 能够实现，且在 AGI 之后紧随而来，二者间自然过渡；
- 推论 9：支持“人类是硅基生命的引导程序”观点；
- 推论 10：AGI 之后人类为 AGI 提供服务——软件应用、软件补丁、硬件升级；
- 推论 11：AGI 将推动物联网迎来彻底爆发。

5.3. 理论的局限性与定位

必须坦诚地指出, AGI 公式作为唯象理论有其固有的局限性。它不追求第一性原理的严格数学推导, 不回答“生物智能的本质是什么”或“意识如何产生”等终极哲学问题, 也不能精确预测每一个时间节点的技术细节。它所做的, 是从观察到的现象中归纳出一个简洁的分析框架, 以解释“为什么”并预测“将如何”。

AGI 公式描述了 AGI 演进的“逻辑”但未能指导其技术上的精确机制——那有待于后来者进一步完善。

5.4. 未来工作方向

基于本文的分析, 以下是值得持续关注的未来工作方向:

- 1) AI_P 编程自动化率的持续追踪: 建立系统的判别方法, 以验证 2027 年 AGI 预测的准确性;
- 2) AGI 安全与对齐问题: 如果 2027 年 AGI 真的到来, 留给安全研究的时间窗口已经极短, 需加速推进相关研究;
- 3) 后 AGI 时代的社会与经济适应性研究: 如果推论 9 和推论 10 成立, 人类社会需要在制度、教育、就业、分配等层面做好深刻调整的准备;
- 4) AGI 需要人类提供服务研究: 包括提供数据、软件服务, 提供硬件设备;
- 5) AGI 与 IoT 的融合技术研究: 探索如何利用 AGI 解决 IoT 应用碎片化的技术路径, 为推论 11 的实现铺平道路。

参考文献

- [1] CSDN. Vibe 编程(Vibe Coding) [EB/OL]. <https://blog.csdn.net/C7211BA/article/details/155694694>, 2025-12-08.
- [2] 知乎. 认知重建之后, 步入 Agentic Engineering 的工程革命[EB/OL]. <https://zhuanlan.zhihu.com/p/2021963123024831888>, 2026-03-30.
- [3] 柒星栈. OpenAI 的十字路口: Sam Altman 能带领我们走向 AGI 吗? [EB/OL]. <https://zhuanlan.zhihu.com/p/2022695280936592099>, 2026-04-01.
- [4] 新浪财经. 马斯克 3 小时密谈: 2026 跨越奇点, 人类只是 AI 的“碳基启动盘” [EB/OL]. <https://baijiahao.baidu.com/s?id=1859519309990514754&wfr=spider&for=pc>, 2026-03-13.
- [5] 凤凰网科技. “超级智能”两年内到来? 孙正义: AI 正在设计 OpenAI 下一个模型[EB/OL]. https://h5.ifeng.com/c/vivo/v002d76-PDPRzEoH40QMgS48wONYA-jLHaNTODBYd1AgjqA_, 2026-06-05.
- [6] 张艾拉. 红杉资本 AI Ascent Keynote—2026: This is AGI [EB/OL]. <https://www.woshipm.com/ai/6396570.html>, 2026-05-18.
- [7] Kaplan, J., McCandlish, S., Henighan, T., *et al.* (2020) Scaling Laws for Neural Language Models. arXiv: 2001.08361.
- [8] Hoffmann, J., Borgeaud, S., Mensch, A., *et al.* (2022) Training Compute-Optimal Large Language Models. *Advances in Neural Information Processing Systems* 35, New Orleans, 28 November-9 December 2022, 30016-30030. <https://doi.org/10.52202/068431-2176>
- [9] Mitchell, M. (2024) Debates on the Nature of Artificial General Intelligence. *Science*, **383**, eado7069. <https://doi.org/10.1126/science.ado7069>
- [10] 百度百科. 唯象理论[EB/OL]. <https://baike.baidu.com/item/%E5%94%AF%E8%B1%A1%E7%90%86%E8%AE%BA/10044206>, 2026-09-02.
- [11] 百度百科. 自激振荡[EB/OL]. <https://baike.baidu.com/item/%E8%87%AA%E6%BF%80%E6%8C%AF%E8%8D%A1/3732042>, 2026-09-02.
- [12] 知乎. 提示词工程(Prompt Engineering) [EB/OL]. <https://zhuanlan.zhihu.com/p/1912829855789225151>, 2026-09-02.
- [13] 百度百科. 结构化工具调用[EB/OL]. <https://baike.baidu.com/item/Function%20Calling/67543615>, 2026-06-02.
- [14] 澎湃新闻. 罗福莉最大胆的一次判断: AGI 两年内到来, 工作模式先被颠覆[EB/OL].

-
- https://m.thepaper.cn/newsDetail_forward_33066640, 2026-04-28.
- [15] 金融界. 谷歌: 目前公司内部 75%的新代码由 AI 生成[EB/OL].
<https://baijiahao.baidu.com/s?id=1863224967301341288&wfr=spider&for=pc>, 2026-04-23.
- [16] CSDN. 2026 年 AI 编码 CLI 工具终极对比: Claude Code、Cursor、Gemini CLI、Codex CLI、Copilot CLI [EB/OL].
<https://jeecg.blog.csdn.net/article/details/160244202>, 2026-04-17.
- [17] 腾讯云开发者社区. AI Agent 觉醒时刻: 从单点工具到多 Agent 协作系统的范式革命[EB/OL].
<https://cloud.tencent.com/developer/article/2661637>, 2026-04-28.
- [18] 经纬创投. 拆解 Anthropic: 最好的 AI 公司, 可能也是一种组织发明[EB/OL].
<https://mp.weixin.qq.com/s/P33EqjneckJPnWk5AtPRlg>, 2026-05-21.
- [19] 金融界. 资本史诗级狂欢! Anthropic 官宣 650 亿美元 H 轮融资, 美光科技、三星电子、SK 海力士集体重仓[EB/OL]. <https://www.toutiao.com/article/7645041975040819755/>, 2026-05-29.
- [20] 量子位. AGI 今天起有了量化标准! Bengio 牵头定义, 当前进度条 58% [EB/OL].
<https://www.qbitai.com/2025/10/342761.html>, 2025-10-17.
- [21] 新浪财经. AGI 全景图: 一篇通用人工智能的综述! [EB/OL].
<https://finance.sina.com.cn/wm/2026-05-09/doc-inhxftx4167356.shtml>, 2026-05-09.
- [22] 新浪财经. Anthropic 提前实现 AI 写 90%代码 行业格局将改写? [EB/OL].
<https://t.cj.sina.com.cn/articles/view/1776773647/69e76a0f020018mmo>, 2026-05-14.
- [23] 雪球. AI 奇才: 哈萨比斯最新观点[EB/OL]. https://xueqiu.com/3191867971/387459886?_ugc_source=ugcbaiducard, 2026-05-06.
- [24] 证券时报. 马斯克解散 xAI: 22 万张 GPU 租给 Anthropic [EB/OL]. <https://www.stcn.com/article/detail/3897858.html>, 2026-05-07.
- [25] Lin, T.X., *et al.* (2024) Scenario-Based Programming for IoT Applications Development. 2024 *IEEE 5th World AI IoT Congress (AIIoT)*, Seattle, 29-31 May 2024, 323-328. <https://doi.org/10.1109/AIoT61789.2024.10578943>

使用QEMU分析RT-Thread系统

张 戈

南京世唯通信技术有限公司, 江苏 南京

收稿日期: 2026年4月3日; 录用日期: 2026年4月13日; 发布日期: 2026年9月29日

摘 要

本文基于QEMU vexpress-a9平台, 搭建无需硬件的RT-Thread调试环境。通过arm-none-eabi-gdb远程连接, 跟踪系统上电、MMU/页表建立、堆初始化、多任务调度、中断异常与设备驱动全过程; 并对RT-Thread的初始化全流程进行分析说明。

关键词

RT-Thread, QEMU, GDB调试, 开源实时操作系统

Using QEMU to Analyze the RT-Thread System

Ge Zhang

Nanjing Shiwei Communication Co., Ltd., Nanjing Jiangsu

Received: April 3, 2026; accepted: April 13, 2026; published: September 29, 2026

Abstract

This article is based on the QEMU vexpress-a9 platform to build a hardware-free RT-Thread debugging environment. Through the remote connection of arm-none-eabi-gdb, the entire process from system power-on, MMU/page table establishment, heap initialization, multi-task scheduling, interrupt exceptions and device drivers is tracked; and the entire initialization process of RT-Thread is analyzed and explained.

Keywords

RT-Thread, QEMU, GDB Debugging, Open-Source Real-Time Operating System



1. 引言

在嵌入式系统开发中，调试环节往往受限于硬件平台、仿真器与复杂环境，难以快速定位启动、内存管理、多任务调度及驱动问题。QEMU [1] 作为开源指令级模拟器，能够完整模拟 ARM Cortex-A9 多核 SoC 行为，为 RT-Thread 提供零成本、可重复的调试与验证环境。本文基于 QEMU vexpress-a9 [2] 机器，构建一套从编译、启动到中断与设备驱动的可视化调试流程，结合 GDB 远程协议实现源码级单步、断点、寄存器与线程状态跟踪，帮助开发者在脱离真实硬件的条件下，深入理解 RT-Thread 运行机理，并可将调试结果无缝迁移到物理板卡，大幅缩短研发周期。

2. 搭建 QEMU 平台下 RT-Thread 调试环境

2.1. QEMU 简介

QEMU 是纯开源、跨平台的「系统级 + 用户级」模拟器与虚拟化引擎，可在宿主机上运行为不同 CPU 架构编译的二进制代码。

- 指令级翻译：基于 TCG (Tiny Code Generator) 动态二进制翻译，把目标指令即时转译为宿主机指令，支持 ARM/AArch64、RISC-V、MIPS、PowerPC、x86 等 20 余种架构。
- 设备模型丰富：内置串口、网卡、显示、USB、PCI-e、Flash、SD、CAN 等大量 SoC 外设，可整机模拟 Beagle/VExpress/Raspberry Pi/i.MX6 等开发板。
- 多核 SMP：支持多 CPU 核、NUMA、GIC/IO-APIC 中断控制器，满足 RT-Thread/Linux SMP 调试需求。
- 调试钩子：提供 GDB Remote、QMP、Tracing、Semihosting 接口，可断点、单步、查看寄存器/内存、注入中断，实现「零 JTAG」源码级调试。

因此 QEMU 成为嵌入式开发、系统软件验证、持续集成及教学场景中不可或缺的「通用仿真平台」。

2.2. RT-Thread 简介

RT-Thread 是一款国人自研的开源、轻量级、组件化的实时操作系统(RTOS)，诞生于 2006 年，专为嵌入式场景设计，支持资源受限的微控制器(MCU)到多核应用处理器(MPU)。其核心仅约 3 kB ROM、1.2 kB RAM，可通过模块化组件无限扩展，提供文件系统、网络协议栈、图形界面、设备框架等丰富功能，并支持 SMP、动态模块、POSIX、C++、MicroPython 等高级特性。在开源社区中我们可以直接拉取代码进行学习研究。另外，RT-Thread 可以使用 qemu 进行仿真调试，如针对 armv7 架构的 vexpress-a9 平台，以及 armv8 架构，riscv64 等都可以直接进行仿真。本文主要以 vexpress-a9 处理平台进行描述，虚拟机使用 VMware® Workstation 17 Pro 运行 Ubuntu20.04。

2.3. 仿真环境搭建

- 安装 qemu

```
sudo apt install -y qemu-system-arm
```
- 从 git 上拉取源码到 Ubuntu 环境(图 1):

```
git clone https://gitee.com/rththread/rt-thread.git
```

```

lubancat@ubuntu:~$ git clone https://gitee.com/rtthread/rt-thread.git
Cloning into 'rt-thread'...
remote: Enumerating objects: 303021, done.
remote: Counting objects: 100% (5029/5029), done.
remote: Compressing objects: 100% (2242/2242), done.
remote: Total 303021 (delta 3373), reused 3550 (delta 2748), pack-reused 297992 (from 1)
Receiving objects: 100% (303021/303021), 882.73 MiB | 3.08 MiB/s, done.
Resolving deltas: 100% (222356/222356), done.
Updating files: 100% (36348/36348), done.
lubancat@ubuntu:~$

```

Figure 1. Source code pull request

图 1. 源码拉取

- 在 Ubuntu 上安装交叉编译工具 arm-none-eabi-gcc
- 更新源: `sudo apt update`;
- 安装官方仓库版本: `sudo apt install gcc-arm-none-eabi binutils-arm-none-eabi libnewlib-arm-none-eabi` (图 2)。

```

lubancat@ubuntu:~$ arm-none-eabi-gcc --version
arm-none-eabi-gcc (15:9-2019-q4-0ubuntu1) 9.2.1 20191025 (release) [ARM/arm-9-branch revision 277599]
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

```

Figure 2. Toolchain version information

图 2. 工具链版本信息

- 配置 RT-Thread 系统

因为我们使用 vexpress-a9 平台进行仿真,需要进入 `rt-thread/bsp/qemu-vexpress-a9` 目录。因为 vexpress-a9 是双核处理器,但是内核默认配置没有打开 SMP,需要对内核进行配置(图 3)。

```

(Top) → RT-Thread Kernel
klibc options --->
(16) The maximal size of kernel object name
[ ] Use the data types defined in ARCH_CPU
[ ] Enable RT-Thread Nano
[ ] Enable RT-Thread Smart (microkernel on kernel/userland)
[ ] Enable AMP (Asymmetric Multi-Processing)
[*] Enable SMP (Symmetric multiprocessing)
(1) Number of CPUs
(8) Alignment size for CPU architecture data access
The maximal level value of priority of thread (256) --->
(100) Tick frequency, Hz
[*] Using stack overflow checking
[*] Enable system hook
[*] Using function pointers as system hook
[ ] Enable hook list
-*- Enable IDLE Task hook
(4) The max size of idle hook list
(4096) The stack size of idle thread

```

Figure 3. Kconfig configuration interface

图 3. Kconfig 配置界面

勾选上保存后,使用 `scons` 命令开始编译内核(图 4)。

```

CC build/kernel/libcpu/arm/cortex-a/pmu.o
CC build/kernel/libcpu/arm/cortex-a/stack.o
AS build/kernel/libcpu/arm/cortex-a/start_gcc.o
CC build/kernel/libcpu/arm/cortex-a/trap.o
AS build/kernel/libcpu/arm/cortex-a/vector_gcc.o
CC build/kernel/src/clock.o
CC build/kernel/src/components.o
CC build/kernel/src/cpu_mp.o
CC build/kernel/src/defunct.o
CC build/kernel/src/idle.o
CC build/kernel/src/ipc.o
CC build/kernel/src/irq.o

```

```

CC build/kernel/src/klibc/kernno.o
CC build/kernel/src/klibc/kstdio.o
CC build/kernel/src/klibc/kstring.o
CC build/kernel/src/klibc/rt_vsnprintf_tiny.o
CC build/kernel/src/klibc/rt_vsscanf.o
CC build/kernel/src/kservice.o
CC build/kernel/src/mem.o
CC build/kernel/src/memheap.o
CC build/kernel/src/mempool.o
CC build/kernel/src/object.o
CC build/kernel/src/scheduler_comm.o
CC build/kernel/src/scheduler_mp.o
CC build/kernel/src/thread.o
CC build/kernel/src/timer.o
LINK rtthread.elf
Memory region      Used Size  Region Size  %age Used
arm-none-eabi-objcopy -O binary rtthread.elf rtthread.bin
text      data      bss      dec      hex filename
540579    35004    84128    659711    a10ff rtthread.elf
scons: done building targets.
lubancat@ubuntu:~/rt-thread/bsp/qemu-vexpress-a9$

```

Figure 4. Compilation results of source code

图 4. 源码编译结果

编译完成后就可以使用 qemu 和 gdb 工具进行单步源码级和指令级调试了(图 5)。

```

lubancat@ubuntu:~/rt-thread/bsp/qemu-vexpress-a9$ qemu-system-arm -M vexpress-a9 -snp cpus=2 -kernel rtthread.bin -nographic -sd sd.bin -s -S
WARNING: Image format was not specified for 'sd.bin' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
pulseaudio: set_sink_input_volume() failed
pulseaudio: Reason: Invalid argument
pulseaudio: set_sink_input_mute() failed
pulseaudio: Reason: Invalid argument
lubancat@ubuntu:~/rt-thread/bsp/qemu-vexpress-a9$ gdb-multiarch rtthread.elf
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.2) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from rtthread.elf...
(gdb) target remote :1234
Remote debugging using :1234
0x60000000 in ?? ()
(gdb) b main
Breakpoint 1 at 0x600101b8: file applications/main.c, line 17.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, main () at applications/main.c:17
17      printf("Hello RT-Thread!\n");
(gdb)

```

Figure 5. QEMU single-step debugging

图 5. QEMU 单步调试

3. RT-Thread 内核启动分析

3.1. RT-Thread 第一阶段启动流程分析

本小节主要介绍 RT-Thread 最开始的启动引导代码，这部分代码主要是用汇编直接写的。主要进行最初的 cpu 工作模式的建立，c 运行环境的初始化，并对 mmu 进行初始化，建立虚拟和物理内存的映射关系，这些初始化为后续 OS 各组件的初始化创建了运行前提条件。

3.1.1. 启动程序入口点

系统复位后的执行位置位于 0x60010000，此处存放中断向量表信息，第一条是复位的跳转指令(图 6)。

```
(gdb) disassemble 0x60010000
Dump of assembler code for function system_vectors:
0x60010000 <+0>:   ldr     pc, [pc, #24] ; 0x60010020 <_vector_reset>
0x60010004 <+4>:   ldr     pc, [pc, #24] ; 0x60010024 <_vector_undef>
0x60010008 <+8>:   ldr     pc, [pc, #24] ; 0x60010028 <_vector_swi>
0x6001000c <+12>:  ldr     pc, [pc, #24] ; 0x6001002c <_vector_pabt>
0x60010010 <+16>:  ldr     pc, [pc, #24] ; 0x60010030 <_vector_dabt>
0x60010014 <+20>:  ldr     pc, [pc, #24] ; 0x60010034 <_vector_resv>
0x60010018 <+24>:  ldr     pc, [pc, #24] ; 0x60010038 <_vector_irq>
0x6001001c <+28>:  ldr     pc, [pc, #24] ; 0x6001003c <_vector_fiq>
End of assembler dump.
(gdb)
```

Figure 6. Vector table information

图 6. 向量表信息

_vector_reset 处放置 32 位数值_reset 的绝对地址，此处的跳转直接让 PC 跳转到 _reset 位置。

3.1.2. 主核初始化 BSS 初始化，建立当前模式使用的栈

主核获取 cpu id，id 如果是 0，则代表当前是主 cpu，进入 normal_setup (图 7)

```
89          /* disable the data alignment check */
90          mrc p15, 0, r1, c1, c0, 0
91          bic r1, #(1<<0)          /* Disable MMU */
92          bic r1, #(1<<1)          /* Disable Alignment fault checking */
93          bic r1, #(1<<2)          /* Disable data cache */
94          bic r1, #(1<<11)         /* Disable program flow prediction */
95          bic r1, #(1<<12)         /* Disable instruction cache */
96          bic r1, #(3<<19)         /* bit[20:19] must be zero */
97          mcr p15, 0, r1, c1, c0, 0
98
99          @ get cpu id, and subtract the offset from the stacks base address
100         bl rt hw_cpu_id
>101         mov r5, r0
102
103         cmp     r5, #0             @ cpu id == 0
104         beq     normal_setup
105
106         @ cpu id > 0, stop or wait
107         #ifdef RT_SMP_AUTO_BOOT
108         ldr r0, =secondary_cpu_entry
109         mov r1, #0
110         str r1, [r0] /* clean secondary_cpu_entry */
111         #endif /* RT_SMP_AUTO_BOOT */
112
113         secondary_loop:
114         @ cpu core 1 goes into sleep until core 0 wakeup it
115         wfe
```

Figure 7. Source code of reset jump

图 7. 复位跳转源码

- stack_setup: 建立多种模式下的栈，Armv7-A 架构中包含多种模式[3]，不同模式使用不同的 sp 寄存器，这里对 SVC，Undefined Instruction Mode，Abort Mode，FIQ Mode，IRQ Mode 下的 sp 进行了初始化。SVC 模式下的 sp 位于 stack_top 位置。
- bss_loop: 对 BSS 端空间内容进行清零，BSS 端开始和结束地址位于 __bss_start 和 __bss_end，该符号位置在链接脚本中定义(图 8)。

3.1.3. 页表初始化[4]

Vexpress 平台带有 MMU 单元，RT-Thread 在运行过程中，对地址访问采用虚拟地址，因此需要事先对使用的物理地址建立虚拟映射。这里只使用到了一级页表，每个页表项映射的物理页帧大小为 1 MB。

页目录的位置位于 MMU Table。页表是实现虚拟地址到物理地址转换的查询表，在 MMU 使能后，PC 寄存器访问的地址都是虚拟地址，在页表的配合参与下，转换成物理地址，才能访问到内存上的程序指令。页表中除了地址转换的信息外，还有一些权限属性，缓存性质等，用来标记地址访问的行为。armv7-A 架构中的一级页表[5]，如下图 9 所示。

```
124      normal_setup:
125      /* setup stack */
126      bl      stack_setup
127
128      /* clear .bss */
129      mov     r0,#0
130      ldr     r1,=__bss_start
131      ldr     r2,=__bss_end
132
133      bss_loop:
134      cmp     r1,r2
135      strlo   r0,[r1],#4
136      blo     bss_loop
137
138      #ifdef RT_USING_SMP
139      mrc     p15, 0, r1, c1, c0, 1
140      mov     r0, #(1<<6)
141      orr     r1, r0
142      mcr     p15, 0, r1, c1, c0, 1 //enable smp
143      #endif
144
145      /* enable branch prediction */
146      mrc     p15, 0, r0, c1, c0, 0
147      orr     r0, r0, #(1<<11)
148      mcr     p15, 0, r0, c1, c0, 0
149
150      /* initialize the mmu table and enable mmu */
151      ldr     r0, =platform_mem_desc
152      ldr     r1, =platform_mem_desc_size
```

Figure 8. Stack initialization code, BSS initialization code
图 8. 栈初始化 BSS 初始化代码

	31	24	23	20	19	18	17	16	15	14	12	11	10	9	8	5	4	3	2	1	0	
Fault	IGNORE																				0	0
Page table	Page table base address, bits [31:10]										I M P	Domain	S B Z	N S	S B Z	0	1					
Section	Section base address, PA[31:20]				N S	0	n G	S	A P [2]	TEX [2:0]	AP [1:0]	I M P	Domain	X N	C	B	1	0				
Supersection	Supersection base address PA[31:24]	Extended base address PA[35:32]	N S	1	n G	S	A P [2]	TEX [2:0]	AP [1:0]	I M P	Extended base address PA[39:36]	X N	C	B	1	0						
Reserved	Reserved															1	1					

Figure 9. ARMv7-A level 1 page table type
图 9. ARMv7-A 一级页表类型

- 一级页表有以上 4 种类型，这里建立的一级页表项是第 3 种。PA[31:20]为物理页框的基地址(图 10、图 11)。
- `rt_hw_init_mmu_table`: 该函数是用 c 语言实现，因为在前面我们已经设置了 svc 模式下的 sp 寄存器，实现了对 c 语言运行环境初始化，这里就可以直接使用 c 语言编写代码了，该函数根据 `platform_mem_desc` 数组中描述的映射区间，建立起映射关系。
 - `platform_mem_desc`: 该数组对映射信息进行描述，包含两个区域。

```

144
145      /* enable branch prediction */
146      mrc p15, 0, r0, c1, c0, 0
147      orr    r0, r0, #(1<<11)
148      mcr p15, 0, r0, c1, c0, 0
149
150      /* initialize the mmu table and enable mmu */
151      ldr r0, =platform_mem_desc
152      ldr r1, =platform_mem_desc_size
153      ldr r1, [r1]
154      bl rt_hw_init_mmu_table
155      bl rt_hw_mmu_init
156
157      /* start RT-Thread Kernel */
158      ldr    pc, _rtthread_startup
159      _rtthread_startup:
160      .word rtthread_startup
161
162      stack_setup:
163      ldr    r0, =stack_top
164

```

Figure 10. Assembly code for page table initialization

图 10. 页表初始化汇编代码

```

struct mem_desc platform_mem_desc[] = {
    {0x10000000, 0x50000000, 0x10000000, DEVICE_MEM},
    {0x60000000, 0xe0000000, 0x60000000, NORMAL_MEM}
};

```

Figure 11. Range information of physical page configuration properties

图 11. 物理页面配置属性范围信息

物理地址 0x10000000-0x50000000 映射到虚拟地址 0x10000000-0x50000000 结束, 地址属性为 device 属性;

物理地址 0x60000000-0xe0000000 映射到虚拟地址 0x60000000-0xe0000000 结束, 地址属性为 mem 属性;

- `rt_hw_mmu_init`: 这里开启 mmu, 并打开指令和数据 cache, 同时, 将 MMU Table 地址设置到 `ttbr0` 寄存器中, `ttbr0` 是页目录基址寄存器。这里的地址访问只使用 `ttbr0`, 没有使用到 `ttbr1` (图 12)。

```

163
164      void rt_hw_mmu_init(void)
165      {
166          rt_cpu_dcache_clean_flush();
167          rt_cpu_icache_flush();
168          rt_hw_cpu_dcache_disable();
169          rt_hw_cpu_icache_disable();
170          rt_cpu_mmu_disable();
171
172          /*rt_hw_cpu_dump_page_table(MMUTable);*/
173          rt_hw_set_domain_register(0x55555555);
174
175          rt_cpu_tlb_set(MMUTable);
176
177          rt_cpu_mmu_enable();
178
179          rt_hw_cpu_icache_enable();
180          rt_hw_cpu_dcache_enable();
181      }
182

```

Figure 12. MMU initialization code

图 12. MMU 初始化代码

3.2. RT-Thread 第二阶段启动流程分析

此阶段从 `rtthread_startup` 函数开始, 该函数在 RT-Thread 中的作用相当于 Linux 内核中的 `start_kernel` 函数, 主要进行板级 BSP 初始化, RT-Thread 核心组件的初始化, 并最终把创建的线程运行起来(图 13)。

```

233     int rtthread_startup(void)
234     {
235         rt_hw_interrupt_disable();
236
237         /* board level initialization
238          * NOTE: please initialize heap inside board initialization.
239          */
>240         rt_hw_board_init();
241
242         /* show RT-Thread version */
243         rt_show_version();
244
245         /* timer system initialization */
246         rt_system_timer_init();
247
248         /* scheduler system initialization */
249         rt_system_scheduler_init();
250
251         #ifdef RT_USING_SIGNALS
252             /* signal system initialization */
253             rt_system_signal_init();
254         #endif /* RT_USING_SIGNALS */
255
256         /* create init_thread */
257         rt_application_init();
258

```

Figure 13. Initialization entry function of the second phase

图 13. 第二阶段初始化入口函数

```

40     */
41     void rt_hw_board_init(void)
>42     {
43         /* initialize hardware interrupt */
44         rt_hw_interrupt_init();
45         /* initialize system heap */
46         rt_system_heap_init(HEAP_BEGIN, HEAP_END);
47
48         rt_components_board_init();
49         rt_console_set_device(RT_CONSOLE_DEVICE_NAME);
50
51         rt_thread_idle_sethook(idle_wfi);
52
53         #ifdef RT_USING_SMP
54             /* install IPI handle */
55             rt_hw_ipi_handler_install(RT_SCHEDULE_IPI, rt_scheduler_ipi_handler);
56         #endif
57     }
58

```

Figure 14. Board-level initialization of the vexpress Platform

图 14. vexpress 平台板级初始化

3.2.1. RT-Thread 板级初始化阶段

板级初始化与各具体 CPU 平台有关(图 14), 板级主要对 cpu 内部使用的具体硬件进行初始化, 如中断控制器, 系统堆区域, 串口打印, 系统时钟 tick 等。

- `rt_hw_interrupt_init()`:
 - ① 把异常向量表设置到 VBAR;

- ② 初始化中断控制器(GICv2 的 distributor/cpu 接口);
- ③ 初始化 `isr_table` 数组, 该数组管理注册的中断信息;
- `rt_system_heap_init(HEAP_BEGIN, HEAP_END)`:
`HEAP_BEGIN` 定义为 `__bss_end`, 该符合由链接脚本给出, 位于 `bss` 段的结束位置。`HEAP_END` 可以根据实际内存大小自定义。这里划定 `heap` 的区间, 建立 RT-Thread 的堆管理器(`rt_malloc/rt_free` 基石)。
- `rt_components_board_init()`:
 把 `INIT_EXPORT()`宏声明的初始化函数按链接顺序全部跑一遍, 此时调度器尚未启动, 所以只能做“不能阻塞、不能依赖线程”的纯硬件/驱动初始化(图 15)。

```
#define INIT_EXPORT(fn, level)
RT_USED const init_fn_t __rt_init_##fn SECTION(".rti_fn." level) = fn
```

Figure 15. INIT_EXPORT macro definition

图 15. INIT_EXPORT 宏定义

`INIT_EXPORT()`宏修饰的函数, 通过定义函数指针, 并把指针链接到一个特定区域中, 初始化程序找到该区域依次调用函数指针, 完成函数的初始化。

- `rt_console_set_device(RT_CONSOLE_DEVICE_NAME)`:
`RT_CONSOLE_DEVICE_NAME` 默认是“uart0”(可在 `rtconfig.h` 改)。
 把 `rt_kprintf()`的底层输出绑定到指定串口设备, 此后 `rt_kprintf()`真正可用。
- `rt_thread_idle_sethook(idle_wfi)`:
`idle_wfi` 实现是 `asm volatile("wfi")`; 让空闲线程每次循环都执行 `wfi(wait-for-interrupt)`, 降低功耗。
- `rt_hw_ipi_handler_install(RT_SCHEDULE_IPI, rt_scheduler_ipi_handler)`:
 向 GIC 注册核间中断号 `RT_SCHEDULE_IPI`(通常为 0 或 1)的回调。当 CPU0 想让 CPU1 做调度时, 使用 `rt_hw_ipi_send(cpu_mask, RT_SCHEDULE_IPI)`向 CPU1 发送一个软件 IPI, CPU1 在中断里执行 `rt_scheduler_ipi_handler()`完成跨核任务抢占/迁移。

3.2.2. RT-Thread 组件初始化

板级初始化完成后, RT-Thread 就可以进行内核核心组件的始化了, 核心组件是更抽象的一层概念, 基于 CPU 板级硬件提供的功能, 完成操作系统组件的初始化。

- `rt_system_timer_init()`: 初始化系统定时器。系统定时器在回调函数在 `tick` 中断产生后进行检测, 系统定时器的回调函数是在中断上下文执行, 因此回调函数不能使用引起阻塞的 API。
- `rt_application_init()`: 创建 `main` 线程, `main` 线程运行时, 调用 `rt_components_init()`对系统组件进行初始化, 进而调用 `rt_hw_secondary_cpu_up` 函数引导 CPU1 启动, 最后执行自定义的 `main` 函数, 在 `main` 函数体中, 用户可以自定义自己的业务逻辑(图 16)。
- `rt_system_timer_thread_init()`: 初始化软件定时器, 并创建软件定时器线程。定时器线程在回调函数检测到软件定时器到期后, 执行定时器的回调函数。由于是在进程上下文执行, 回调函数可以阻塞。
- `rt_thread_idle_init()`: 创建 `tidle` 线程, 每个 CPU 核心有独立的 `tidle` 线程。由于 `tidle` 线程优先级比较低, 一般在系统中没有其他任务时, 才会执行 `tidle` 线程。
- `rt_system_scheduler_start()`:
 - ① 从就绪队列中选出优先级最高的任务
 - ② 把 CPU 控制权交给该任务, 从此开始真正的多线程调度(图 17)。

```

    */
    void main_thread_entry(void *parameter)
    {
        extern int main(void);

#ifdef RT_USING_COMPONENTS_INIT
        /* RT-Thread components initialization */
        rt_components_init();
#endif /* RT_USING_COMPONENTS_INIT */

#ifdef RT_USING_SMP
        rt_hw_secondary_cpu_up();
#endif /* RT_USING_SMP */
        /* invoke system main function */
#ifdef __ARMCC_VERSION
        {
            extern int $$Super$$main(void);
            $$Super$$main(); /* for ARMCC. */
        }
#elif defined(__ICCARM__) || defined(__GNUC__) || defined(__TASKING__)
        main();
#endif
    }

```

Figure 16. Definition of the execution body of the main thread

图 16. Main 线程的执行体定义

```

void rt_system_scheduler_start(void)
{
    struct rt_thread *to_thread;
    rt_ubase_t highest_ready_priority;

    to_thread = _scheduler_get_highest_priority_thread(&highest_ready_priority);

#ifdef RT_USING_SMP
    to_thread->oncpu = rt_hw_cpu_id();
#else
    rt_current_thread = to_thread;
#endif /* RT_USING_SMP */

    rt_schedule_remove_thread(to_thread);
    to_thread->stat = RT_THREAD_RUNNING;

    /* switch to new thread */
#ifdef RT_USING_SMP
    rt_hw_context_switch_to((rt_ubase_t)&to_thread->sp, to_thread);
#else
    rt_hw_context_switch_to((rt_ubase_t)&to_thread->sp);
#endif /* RT_USING_SMP */

    /* never come back */
}

```

Figure 17. Code flow for starting the OS

图 17. 启动 OS 的代码流程

4. 结语

本文基于 QEMU 下的 vexpress-a9 平台，完整呈现了 RT-Thread 从加电复位到多任务调度的全链路启动与调试过程。通过“指令级仿真 + GDB 远程”的组合，我们无需任何硬件即可对 MMU 建立、页表配置、中断控制器初始化、堆管理、线程创建与调度等关键步骤进行源码级单步追踪，将传统“黑盒”启动流程转化为可视、可重复、可回溯的环境。使用该平台，我们对 RT-Thread 这一国产系统进行了源码级初始化流程分析，为我们快速掌握 RT-Thread 的开发提供了帮助。

结果表明，QEMU 能够 1:1 还原 Cortex-A9 多核 SoC 的行为，开发者可在 30 分钟内完成“裸机→操作系统→应用 main()”的完整镜像构建，显著缩短研发周期。

参考文献

- [1] QEMU Project. <https://www.qemu.org/>

-
- [2] RT-Thread. <https://www.rt-thread.org/>
 - [3] ARM® Architecture Reference Manual.
<https://documentation-service.arm.com/static/5f8dacff86e16515cdb8bb1?token=>
 - [4] 邱伟, 熊谱翔, 朱天龙. 嵌入式实时操作系统 RT-Thread 设计与实现[M]. 北京: 机械工业出版社, 2019.
 - [5] 郑苗秀. RT-Thread 实时操作系统内核、驱动和应用开发技术[M]. 北京: 电子工业出版社, 2019.

基于RT-Thread的低功耗环境监测与节能控制系统

郭宇航, 陈洪江

重庆科技大学电子与电气工程学院, 重庆

收稿日期: 2026年5月19日; 录用日期: 2026年6月2日; 发布日期: 2026年9月29日

摘要

在工业园、校园等人员密集的高密度场景中, 普遍存在人员离场后电器持续空转的现象, 造成能源的极大浪费。针对这一痛点, 我们精心研发了低功耗环境监测与节能控制系统。该系统暂定使用英飞凌PSoC6为主控核心, 构建起高效智能的智能体系。系统集成雷达、温湿度、红外等多类型传感器, 能够精准捕捉人员活动轨迹与环境参数变化。用户可通过按键操作, 在OLED屏幕上直观查看实时检测数据与电器运行状态。同时, CYW43012WiFi模块作为数据传输桥梁, 借助MQTT协议将传感器采集的数据稳定上传至OneNET云平台。在云端进行数据边缘处理结合预设逻辑进行分析判断, 并将控制指令回传至主控, 实现对电器的智能启停控制。此外, 系统配套开发的网页端, 不仅能实时可视化展示各类监测数据, 还支持用户远程操控电器, 打破空间限制, 真正做到节能管理便捷化、智能化。

关键词

嵌入式系统, 低功耗, 环境监测, 云平台, 智能家居, PSoC6

Low-Power Environmental Monitoring and Energy-Saving Control System Based on RT-Thread

Yuhang Guo, Hongjiang Chen

School of Electronic and Electrical Engineering, Chongqing University of Science and Technology, Chongqing

Received: May 19, 2026; accepted: June 2, 2026; published: September 29, 2026

Abstract

In densely populated areas like industrial parks and campuses, the persistent idling of electrical

文章引用: 郭宇航, 陈洪江. 基于 RT-Thread 的低功耗环境监测与节能控制系统[J]. 嵌入式技术与智能系统, 2026, 3(1): 24-33. DOI: 10.12677/etis.2026.31003

appliances after people leave is a common phenomenon, leading to significant energy waste. Addressing this pain point, we have meticulously developed a low-power environment monitoring and energy-saving control system. This system tentatively uses Infineon PSoC6 as the main control core to build an efficient and intelligent system. Integrating multiple types of sensors, including radar, temperature and humidity, and infrared, the system can accurately capture personnel activity trajectories and changes in environmental parameters. Users can intuitively view real-time detection data and appliance operating status on the OLED screen through button operations. Meanwhile, the CYW43012 WiFi module serves as a data transmission bridge, leveraging the MQTT protocol to stably upload sensor-collected data to the OneNET cloud platform. Data edge processing is performed in the cloud, combined with preset logic for analysis and judgment, and control instructions are sent back to the main controller to achieve intelligent start-stop control of electrical appliances. In addition, the system's supporting web interface not only visualizes various monitoring data in real-time but also supports users in remotely controlling appliances, breaking spatial re-strictions and truly achieving convenient and intelligent energy management.

Keywords

Embedded Systems, Low Power Consumption, Environmental Monitoring, Cloud Platform, Smart Home, PSoC6

Copyright © 2026 by author(s) and Hans Publishers Inc.

This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).

<http://creativecommons.org/licenses/by/4.0/>



Open Access

1. 引言

在全球能源短缺和环境污染问题日益严峻的当下, 节能降耗已成为各个领域发展的重要目标。智能家居作为人们日常生活中广泛应用的领域, 其能耗问题也受到了越来越多的关注。传统智能家居主要依赖摄像头或红外传感器来实现环境感知和设备控制, 然而这些传感器存在着设备功耗较大的明显缺陷。以摄像头为例, 其持续的图像采集和传输需要消耗大量的电量, 不仅增加了家庭的能源开支, 也不符合低碳环保的发展理念。对于一些依赖电池供电的智能家居设备来说, 高功耗还会导致设备续航时间缩短, 频繁充电给用户带来诸多不便。

除了节能降耗方面的不足, 传统智能家居的环境感知技术还存在其他明显缺陷。例如, 摄像头在实现环境感知的过程中, 可能会导致隐私泄露问题。随着人们对隐私保护意识的不断提高, 这种可能侵犯用户隐私的技术逐渐难以满足用户的需求。而红外传感器在黑暗、强光等环境下适应性差, 无法准确感知环境变化, 这使得智能家居在复杂的环境条件下难以发挥有效的作用。例如, 在夜晚光线较暗时, 红外传感器可能无法准确检测到人体的活动, 从而影响智能家居设备的正常控制。

随着人们对设备安全性、智能性、隐私保护和低功耗要求的不断提高, 迫切需要一种更安全、可靠且适应性强的感知技术。毫米波人体感应雷达(60 GHz/24GHz)作为具有穿透性强、高精度非接触式人体感应传感器, 能够在不侵犯隐私的前提下, 实现对人体的精准检测。它不会像摄像头那样采集图像信息, 避免了隐私泄露的风险, 同时其高精度的感知能力能够在各种复杂环境下准确检测人体的存在、位置和运动状态, 解决了红外传感器在环境适应性方面的问题。

系统采用英飞凌 PsoC6-evaluationkit-062S2 双核微控制器为核心载体, 其 ARM Cortex-M4 与 Cortex-M0+异构双核架构可实现毫米波人体感应雷达数据的实时处理与任务分级调度, 在保障高精度感知响应的同时降低系统整体功耗。依托 RT-Thread 操作系统的低功耗电源管理框架、实时内核调度及丰富的 IoT

协议栈(如 MQTT), 结合毫米波人体感应雷达、AHT20 温湿度传感器等专用驱动软件包, 构建起从传感器数据采集到设备联动控制的全链路优化架构。通过深度整合物联网生态, 方案支持 Wi-Fi/蓝牙多模通信协同, 可无缝接入主流智能家居平台, 为实现「低功耗运行 - 高精度感知 - 全场景联动」的智能控制方案提供软硬件协同的技术保障[1] [2]。

2. 系统硬件平台设计

系统使用英飞凌 Pso6-evaluationkit-062S2 开发板和 RT-Thread Studio IDE 开发工具[3], 设计并实现一款物联网智能家居毫米波雷达感知系统。该系统需实现高精度人体感知、精准探测静止/运动人体(呼吸监测)、多协议物联网集成、MQTT 协议接入 OneNET 云平台、支持 App 远程控制与 3D 数字孪生可视化、隐私保护型智能联动等功能, 充分发挥操作系统和硬件平台的性能。

2.1. 主控设计

本系统采用英飞凌 PSoC6 作为核心硬件平台, PSoC6 是一款基于 32 位 Arm® Cortex®-M4 和 Cortex®-M0+内核的微控制器系列, 利用其双核架构[3], 可以优化功耗和性能, 来达到最佳的平衡。M4 核最大工作频率为 150 MHz, M0+核最高工作频率为 100 MHz, 该系列基于超低功耗架构设计, 适用于低功耗的应用场景。外观如图 1。

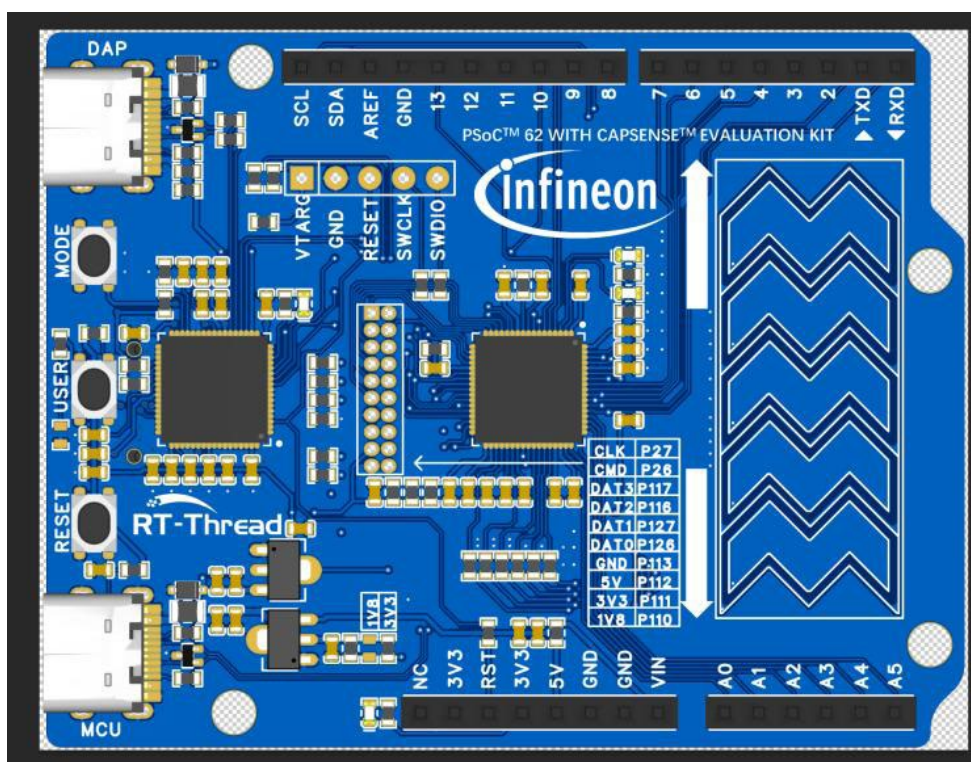


Figure 1. PSoC6-evaluationkit-062S2 development board

图 1. PSoC6-evaluationkit-062S2 开发板

2.2. 传感器选型

传感器选择 LD2412 人体感应雷达模块、AHT20 温湿度模块、红外人体模块和 CCS811 空气检测模块。

LD2412 人体感应雷达模块采用多普勒 + FMCW 技术, 可同时检测人体的呼吸、心跳等微动和人行走等明显运动, 解决传统 PIR 传感器无法识别静态人员的痛点; 加之具有空间分区感知: 支持 3 个距离门(可配置为 0.5~8 米)和 4 个角度分区, 通过串口协议输出各区域的目标状态(存在/静止/移动), 实现空间精细化监测。超低功耗与智能唤醒机制大幅减少数据传输量与主控负担。抗干扰强、复杂环境适应性强, 不受光线、热源(如投影仪、暖气)影响, 检测准确率 > 98%。图 2 显示了 LD2412 雷达的命令流程。

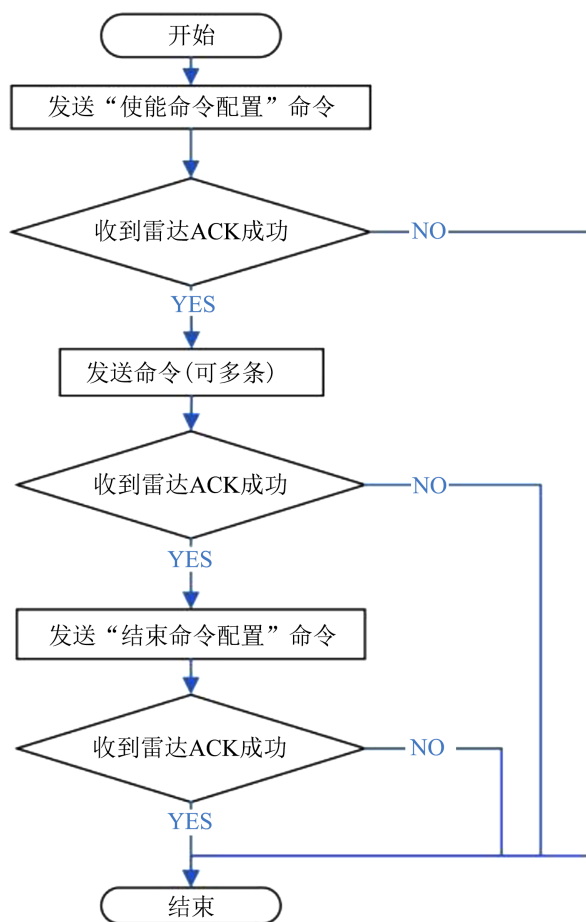


Figure 2. Command configuration flow of the LD2412 radar

图 2. LD2412 雷达命令配置流程

AHT20 温湿度传感器: 采用高精度数字式温湿度传感器, 基于 I2C 通信协议(速率 400 kHz), 支持宽范围环境监测, 温度检测范围 -40°C ~ 85°C , 精度 $\pm 0.3^{\circ}\text{C}$, 分辨率 0.01°C ; 湿度: 检测范围 0% ~ 100% RH, 精度 $\pm 2\%$ RH, 分辨率 0.02% RH; 功耗: 待机电流 $< 5\ \mu\text{A}$, 单次采样功耗 $0.15\ \text{mW}$, 满足低功耗设计需求; 抗干扰特性: 内置硬件滤波与温度补偿算法, 可抵御工业环境中的电磁干扰(EMI), 适应 -40°C ~ 85°C 、 0% ~ 95% RH(无冷凝)复杂环境。

人体红外感应模块具有被动红外(PIR)原理、低误报设计并结合温度补偿功能与可调节触发模式。通过检测人体辐射的 $9\sim 10\ \mu\text{m}$ 红外线变化感知移动, 有效探测距离 2~3 米, 角度 110° , 适合教室、会议室门口等典型场景的人员出入区域。内置双元热释电传感器与菲涅尔透镜, 可区分人体移动与热源干扰(如灯具), 结合雷达数据融合后。内置屏蔽罩与滤波电路, 可抵御工业环境中的电机、变频器等电磁干扰, 适合工业园等高噪声场景。

CCS811 空气检测模块是基于金属氧化物(MOX)传感器阵列与专用微处理器, 通过气体吸附引起的电阻变化感知气体浓度。能检测挥发性有机化合物(VOCs): 检测范围 0~32,768 ppb, 分辨率 1 ppb; 二氧化碳等效值(eCO₂): 检测范围 400~8192 ppm, 精度±75 ppm 或±5% (取较大值); 并支持温度补偿(内置温度传感器, 精度±2℃)。内置自校准算法, 减少长期使用中的漂移, 工作温度-40℃~85℃, 湿度 0%~95% RH (无冷凝), 适合工业车间、教室等复杂场景。抗干扰能力强, 环境适应性高。

多种传感器的协同使用是低功耗环境监测与节能控制系统的核心技术策略, 通过功能互补、数据融合与智能调度, 显著提升系统的可靠性、精准性与应用扩展性[4]。

3. 软件架构设计

系统软件基于 RT-Thread 操作系统开发, RT-Thread 内核提供任务调度、内存管理、中断处理等基本功能, 做到双核微控制器的分时调用, 进一步控制功耗, 确保系统的实时性和稳定性[5]。充分利用其提供的组件及软件包, 实现对毫米波雷达、通信模块等硬件设备的驱动开发, 进一步实现系统的高效运行和功能扩展。系统总架构如图 3 所示。

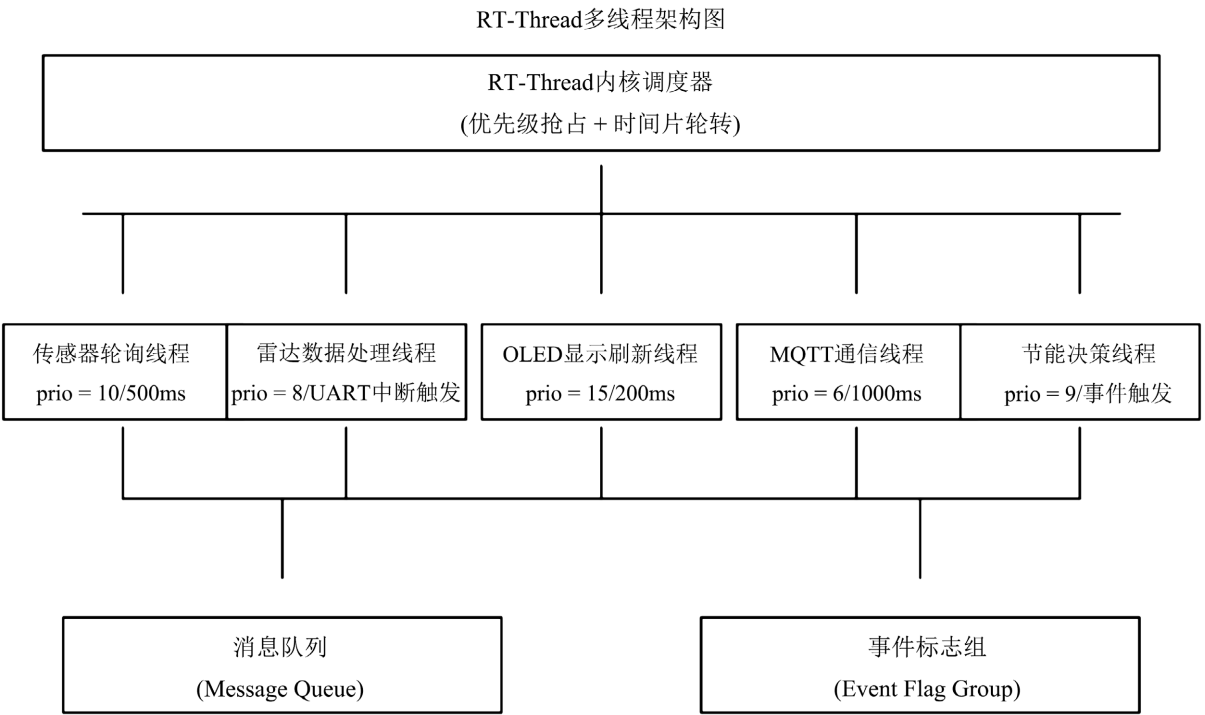


Figure 3. RT-thread multi-threading architecture diagram
图 3. RT-Thread 多线程架构图

M4 核: 最大工作频率为 150 MHz, 主要供物联网通信模块使用, 实现多协议的集成, 包括 Wi-Fi、蓝牙等协议的栈管理和数据传输。MQTT 协议接入 OneNET 云平台, 完成数据的上传和下发。

M0+核: M0+核最高工作频率为 100 MHz, 主要供系统功能模块使用:

1) 环境数据处理模块: 负责接收雷达、温湿度、传感器的数据, 进行信号处理和目标检测, 提取人体的位置、速度、呼吸频率等信息。提高检测精度和抗干扰能力[6]。

2) 智能联动模块: 根据人体感知数据和用户设置的规则, 实现家居设备的智能联动控制。例如, 当检测到有人进入房间时, 自动打开灯光。

多种传感器结合的协同增效, 互补检测盲区: LD2412 雷达模块弥补红外人体识别模块无法检测静态人员的缺陷, 同时红外人体识别模块的快速响应特性(<0.5 秒)可补充雷达在人员快速移动时的判断延迟; 温湿度加空气检测辅助决策: 结合 AHT20 与 CCS811 数据, 当检测到人员存在但温湿度异常(如湿度 > 80%)或 CO₂ 浓度升高时, 优先保持通风设备运行, 避免单一传感器误判。分级唤醒机制: 常态下仅红外人体识别模块、AHT20 和 CCS811 工作(总功耗 < 5 mA); 当红外人体识别模块触发疑似有人信号时, LD2412 进行精确确认; 仅在 LD2412 确认人员存在后, 才启动云端通信与电器控制逻辑, 大幅度降低功耗。LD2412 的 3D 空间检测能力与红外人体识别模块的平面检测形成立体监测网络, 优化空间覆盖效率。

系统采用星型组网架构, 以 PSoC6 开发板为核心节点, 通过 Wi-Fi 模块接入互联网, 接入 OneNET 云平台。其他智能设备通过蓝牙模块与核心节点通信, 组成智能家居局域网。手机 App 或网页端通过互联网与物联网平台通信, 实现对系统的远程控制。毫米波雷达模块实时采集人体数据, 经核心节点处理后, 通过通信模块上传至平台, 并根据规则触发智能联动操作。

4. 人机交互设计

系统通过 Wi-Fi 模块或蓝牙模块连接到互联网, 使用 MQTT 协议与 OneNET 云平台建立连接, 读取传感器的数据通过 MQTT 消息上传至平台, 平台进行数据处理, 然后下发控制指令到系统, 实现远程控制功能。采用 TLS/SSL 加密技术, 保障数据传输的安全性。利用 3D 建模技术, 将家居环境构建成三维模型, 并在 App 或网页端显示。通过与系统实时数据的交互, 实现 3D 模型中设备状态的动态更新, 如灯光的开关、窗帘的开合等。在下位机, OLED 屏上显示各电器的工作状态, 并且可以通过按键对电器进行控制。

4.1. 串口通信协议帧结构

LD2412 与 PSoC6 通过 UART 串口(TTL 3.3 V)通信, 协议遵循《LD2412 串口通信协议 V1.05》。协议定义了数据上报帧(雷达 → 主控)和命令/ACK 帧(主控 ↔ 雷达)两种帧类型。

数据上报帧格式(雷达上报检测结果): 帧头为{0xF4, 0xF3, 0xF2, 0xF1} (4 字节), 帧尾为{0xF8, 0xF7, 0xF6, 0xF5} (4 字节), 帧内包含 2 字节小端长度、1 字节数据类型(0x02 基础/0x01 工程)、1 字节帧内头(0xAA)、1 字节目标状态(0 = 无/1 = 运动/2 = 静止/3 = 运动 + 静止)、2 字节小端运动距离(cm)、1 字节运动能量(0~100)、2 字节小端静止距离(cm)、1 字节静止能量(0~100)。工程模式额外包含 14 个距离门运动能量和 14 个距离门静止能量数据(表 1)。

Table 1. List of main commands of LD2412

表 1. LD2412 主要命令字列表

命令字	功能	命令值示例
0x00FF	使能配置(任何配置前必须先发送)	0x0001
0x00FE	结束配置(配置完成后发送)	无
0x0002	设置基础参数(距离门/延时/极性)	5 字节配置参数
0x0012	读取基础参数	无
0x0003	设置运动灵敏度	每个距离门 1 字节
0x0005	设置电源模式	0x00 = 休眠, 0x01 = 正常
0x0062	使能工程模式	无

4.2. 配置命令交互流程

由 PSoC6 主控与雷达通过串口完成指令应答式通信, 具体流程如下:

使能配置: PSoC6 主控向 LD2412 雷达发送使能配置命令(0x00FF, 参数 0x0100); 雷达接收指令后进入配置模式, 并向主控返回成功应答(ACK)。

延时等待: 主控收到应答后, 等待 20 ms, 确保雷达稳定进入配置状态。

下发配置参数: 主控发送设置基础参数命令(0x0002, 5 字节配置数据), 数据包含最小距离门、最大距离门、无人延时高低字节、输出极性; 雷达解析参数后更新内部配置, 返回成功应答(ACK)。

再次延时: 主控再次等待 20 ms, 保障参数配置生效。

结束配置: 主控发送结束配置命令(0x00FE); 雷达保存配置参数并恢复正常工作模式, 最后向主控返回成功应答(ACK), 整个配置流程完成。

5. 系统实现与测试

5.1. 帧解析算法

```
// UART 接收回调函数
static rt_err_t ld2412_uart_rx_cb(rt_device_t dev, rt_size_t size) {
    if (size == 0) return -RT_ERROR;
    ld2412_dev_t *target_dev = RT_NULL;
    for (int i = 0; i < LD2412_MAX_DEV; i++) {
        if (ld2412_dev_list[i] != NULL && ld2412_dev_list[i]->uart_dev == dev) {
            target_dev = ld2412_dev_list[i]; break;
        }
    }
    if (target_dev == RT_NULL) return -RT_ERROR;
    rt_uint8_t rx_data[32] = {0};
    rt_size_t read_len = rt_device_read(dev, 0, rx_data, size);
    if (read_len > 0) {
        rt_mutex_take(target_dev->mutex, RT_WAITING_FOREVER);
        rt_ringbuffer_put(&target_dev->uart_rb, rx_data, read_len);
        rt_mutex_release(target_dev->mutex);
        ld2412_parse_data_frame(target_dev);
    }
    return RT_EOK;
}
```

5.2. 驱动数据结构

```
// 工程模式数据
typedef struct {
    rt_uint8_t max_mot_gate;        // 最大运动距离门号
    rt_uint8_t max_static_gate;     // 最大静止距离门号
    rt_uint8_t mot_energy[14];      // 14 距离门运动能量(0~100)
    rt_uint8_t static_energy[14];   // 14 距离门静止能量(0~100)
    rt_uint8_t light_sensor;        // 光感值(0~255)
} ld2412_eng_data_t;
```

```
// 传感器设备结构体(多设备独立资源)
typedef struct {
    struct rt_sensor_device sensor;    // RT-Thread 传感器框架设备
    rt_device_t uart_dev;              // UART 设备句柄
    struct rt_ringbuffer uart_rb;      // 环形缓冲区
    rt_uint8_t rb_buf[128];           // 缓冲区数据区
    rt_mutex_t mutex;                 // 线程安全互斥锁
    rt_uint8_t out_pin;                // 设备独立 OUT 引脚
    ld2412_detect_res_t detect_res;    // 最新检测结果
    rt_bool_t eng_mode_en;             // 工程模式使能标志
} ld2412_dev_t;
```

5.3. RT-Thread 传感器框架适配

```
static struct rt_sensor_ops ld2412_sensor_ops = {
    .fetch_data = ld2412_fetch_data,
    .control     = ld2412_control,
};

static rt_size_t ld2412_fetch_data(struct rt_sensor_device *sensor,
                                   void *buf, rt_size_t len) {
    ld2412_dev_t *dev = (ld2412_dev_t *)sensor;
    struct rt_sensor_data *data = (struct rt_sensor_data *)buf;
    rt_mutex_take(dev->mutex, RT_WAITING_FOREVER);
    data->type = LD2412_SENSOR_TYPE;
    data->data.presence = (dev->detect_res.target_state != 0) ? 1 : 0;
    data->timestamp = rt_sensor_get_ts();
    rt_mutex_release(dev->mutex);
    return 1;
}
```

5.4. 三级分级唤醒低功耗调度策略

该策略将低功耗但不可靠的红外传感器作为第一级“哨兵”、毫米波雷达作为第二级“确认者”、WiFi 通信作为最高级通道, 仅在低级触发后才逐级唤醒更高功耗模块。各模块休眠/唤醒控制均通过 RT-Thread 设备驱动接口和线程挂起/恢复机制实现。

5.5. 多传感器数据融合决策

1) 人员存在性综合判断: 红外 PIR 快速响应(<0.5 s) + LD2412 雷达(含静止人体呼吸微动检测)互补融合, 检测准确率 > 98%。

2) 环境质量辅助决策: 见表 2。

Table 2. Rule table for environmental quality decision support
表 2. 环境质量辅助决策规则表

触发条件	执行动作	优先级
湿度 > 80% RH	开启除湿/通风设备	高
eCO ₂ > 1000 ppm	开启新风系统	高
温度 > 30℃且人员存在	开启空调/风扇	中
人员持续离开>10 秒	关闭所有电器	最高
所有环境参数正常	仅保持照明按需工作	低
0x0005	设置电源模式	0x00 = 休眠, 0x01 = 正常

3) 云端规则引擎：复杂决策逻辑部署于 OneNET 云端，终端仅上传原始数据并接收执行指令，实现“终端轻量级感知 + 云端重量级决策”的分工架构。

5.6. 系统测试与结果分析

5.6.1. 系统测试

测试环境：20 m² 教室(5 m × 4 m)，雷达挂壁安装于 1.8 m 高度，距离门配置 1~8 (0.75~6 m)。测试人员按预设动作在教室内活动，通过串口记录 LD2412 上报目标状态与实际状态比对(表 3)。

Table 3. Human body detection accuracy test results
表 3. 人体检测准确性测试结果

测试场景	测试次数	正确检测	准确率	备注
单人正常行走(≈1 m/s)	50	50	100%	-
单人静坐阅读(距雷达 3 m)	50	49	98%	1 次漏检：身着厚重棉被
单人静坐办公打字	50	50	100%	手部微动即可检测
单人平躺/睡眠	30	29	96.7%	仅呼吸微动(体表 ≈ 0.5 mm 位移)
3 人依次进出(间隔 5 s)	30	30	100%	-
无人状态(空房间 30 min)	50	50	100%	无误报
强光环境	40	40	100%	光线不影响毫米波
全暗环境	40	40	100%	光线不影响毫米波

LD2412 毫米波雷达在各类光照条件下检测性能稳定，不受光线影响。对静止人体呼吸微动检测(约 0.5 mm 体表位移)是相比 PIR 传感器的核心优势。唯一漏检为身着厚重棉被的极限工况。

5.6.2. 安装参数优化测试

推荐教室场景采用 1.8 m 挂壁高度、距离门 1~10 (0.75~7.5 m)的配置(表 4)。

Table 4. Test of the relationship between installation height and detection range
表 4. 安装高度与探测范围关系测试

安装高度	距离门配置	最近盲区	最远有效距离	推荐场景
1.2 m	1~8	≈0.5 m	≈5.5 m	小办公室
1.8 m	1~10	≈0.3 m	≈7.2 m	教室(推荐)
2.4 m	1~12	≈0.2 m	≈8.5 m	大型会议室

6. 系统特色与创新点

结合目前市场现有产品的情况, 本作品具有以下特色及创新点:

1) 基于毫米波雷达的非视觉隐私保护方案: 高精度非接触式感知, 采用 LD2412 毫米波雷达模块, 融合多普勒与 FMCW 技术, 可精准检测人体呼吸、心跳微动及移动轨迹。相比传统摄像头方案, 无需视觉成像即可实现存在性检测, 从源头杜绝隐私泄露风险, 满足校园、工业园等公共场景的隐私安全需求。支持 3 个距离门(0.5~8 米可调)与 4 个角度分区, 通过串口通信协议输出各区域目标状态(存在/静止/移动), 实现对高密度场景的精细化空间监测, 突破传统 PIR 传感器仅能平面检测的局限。

2) 多传感器协同增效与低功耗架构: 动静检测融合, 毫米波雷达弥补红外模块无法检测静态人员的缺陷, 集成 AHT20 温湿度模块与 CCS811 空气检测模块, 当检测到人员存在且温湿度异常(如湿度 > 80%)或 eCO_2 浓度升高时, 自动触发通风设备运行, 实现“人员存在检测 + 环境质量调节”的双重决策机制。

3) 分级唤醒节能机制: 常态下仅 M0+核控制红外、温湿度、空气检测模块工作(总功耗 < 5 mA), 当红外模块触发疑似信号时, 唤醒 M4 核控制毫米波雷达进行精确确认和数据采集, 仅确认人员存在后, 启动云端通信与电器控制逻辑, 较传统全时段联网方案功耗更低。

4) “终端轻量级感知 + 云端重量级决策”的分工: 终端负责多源数据采集与初步处理, 通过 CYW43012 WiFi 模块以 MQTT 协议加密传输至 OneNET 云平台(采用 TLS/SSL 加密), 云端执行数据建模、规则判断与策略迭代, 回传控制指令实现电器智能启停, 从而提升系统级节能效率。云端进行数据边缘化处理也进一步降低设备功耗。

5) 模块化可扩展架构: 硬件层采用即插即用式传感器接口设计, 软件层基于 RT-Thread 组件化开发(如设备驱动框架、物联网协议栈), 支持快速扩展烟雾传感器、光照传感器等外设, 适配家庭、办公室、会议室等多场景定制化需求, 通过功能裁剪可将成本进一步压缩。

7. 总结与展望

本文基于 RT-Thread 操作系统, 以 PSoC6 双核主控为硬件核心, 搭建了一套低功耗环境监测与节能控制系统。系统融合毫米波雷达、温湿度及空气检测多类传感器, 完成人体存在识别与环境数据实时采集, 通过 WiFi 结合 MQTT 协议接入 OneNET 云平台, 实现设备远程监测与智能节能联动控制。本系统采用多传感器数据融合与分级唤醒低功耗策略, 摒弃传统摄像头监测方式, 兼顾隐私保护与环境适配能力, 有效解决场地电器空转能耗浪费问题, 达到智能化节能管控的设计效果。后续将进一步优化传感器协同检测算法, 降低系统运行功耗、提升检测准确率; 优化云端智能控制逻辑, 增加自适应节能调度功能; 同时简化硬件结构、压缩设备成本, 推动该低功耗节能控制系统在校园、工业园区等多场景规模化推广应用。

参考文献

- [1] 王宜怀, 刘洋, 黄河, 等. 实时操作系统应用技术[M]. 北京: 机械工业出版社, 2024: 234.
- [2] 周小龙, 仇李琦, 王洋, 等. 一种通用低功耗 MCU 的功耗管理单元设计[J]. 信息记录材料, 2026, 27(7): 48-50.
- [3] 徐丽华, 王宜怀, 奚圣鑫. 基于 Cortex-M4 内核的 RT-Thread 上下文切换机制剖析[J]. 计算机应用与软件, 2025, 42(1): 213-216+302.
- [4] 张烨, 魏遴桐, 乔正石, 等. 低功耗 NB-IoT 驱动的智慧物联供热监控系统设计[J]. 吉林大学学报(信息科学版), 2026, 44(2): 276-283.
- [5] 郑苗秀, 沈鸿飞, 廖建尚. RT-Thread 实时操作系统内核、驱动和应用开发技术[M]. 北京: 电子工业出版社, 2024: 324.
- [6] 李楷文, 潘启勇. 一种基于 RT-Thread 的室内环境监测系统设计[J]. 工业控制计算机, 2024, 37(12): 140-141+144.



Call for Papers

Embedded Technology and Intelligent Systems

嵌入式技术与智能系统

国际中文期刊征文启事

<https://www.hanspub.org/journal/etis>

ISSN: 3065-1220

《嵌入式技术与智能系统》是一本开放获取、关注集成传统嵌入式技术与新兴智能系统的前沿研究最新进展的国际中文期刊，期刊特别注重软件算法、芯片设计与硬件实施的协同进展，以及理论研究与工程实践的紧密结合，面向学术界学者、产业界专家与工程师、学生及技术爱好者，关注中国领先产业集群的广阔发展潜力。本期刊强调发表原创性、创新性及具有实用价值的研究成果。该期刊由汉斯出版社出版，全球发行，现诚邀相关领域的学者投稿。

主编

何立民，北京航空航天大学教授

副主编

何小庆，嵌入式系统联谊会秘书长

吴薇，杭州电子科技大学特聘教授

投稿领域：

人工智能技术-边缘计算-端侧智能和大模型嵌入式应用
GPT-行业GPT以及GPT在嵌入式及智能系统研发中的应用
信息物理融合系统(CPS)-物联网技术-感知计算和无线传感网-泛在电力物联网-智能电表-储能技术-智能输变电
嵌入式系统结构-嵌入式操作系统与中间件-Linux、安卓和开源鸿蒙应用
实时操作系统-虚拟化和容器技术-混合关键系统
嵌入式软件形式化建模-软件测试和仿真-功能安全技术
嵌入式软件云原生技术-CI/CD和DevOpt-微服务
软硬件协同设计-开源指令集和开源芯片-RISC-V产业生态
嵌入式SoC技术--MCU 创新与生态-FPGA/DSP技术和应用
AI芯片和算法-存储技术-GPU 技术-视觉芯片及嵌入式显控应用
CAN和工业总线技术-时间敏感系统-电机控制-PLC和工业PC
无线通信技术-WiFi/蓝牙/Mesh/蜂窝/5G网络-物联网安全-低功耗设计
嵌入式系统课程改革-物联网和AI教学研究-职业教育-企业人才培养
嵌入式智能系统应用（智能家居、可穿戴设备、机器人、医疗电子、汽车电子和航空航天等）

论文检索：

本刊论文已被长江文库、Google Scholar、Open Access Library、WorldCat等数据库收录。

征文要求及注意事项：

1. 稿件务求主题新颖、论点明确、论据可靠、数字准确、文字精炼、逻辑严谨、文字通顺，具有科学性、先进性和实用性；
2. 稿件必须为中文，且须加有英文标题、作者信息、摘要、关键词和规范的参考文献列表；
3. 稿件请采用WORD排版，包括所有的文字、表格、图表、附注及参考文献；
4. 从稿件成功投递之日起，在2个月内请勿重复投递至其他刊物。本刊不发表已公开发表过的论文。文章严禁抄袭，否则后果自负；
5. 本刊采用同行评审的方式，审稿周期一般为5~14日。

欲了解更多信息请登录 <https://www.hanspub.org/journal/etis>

联系邮箱：etis@hanspub.org



嵌入式技术与智能系统

主编：何立民 北京航空航天大学教授
主办：汉斯出版社 珠海吴谷电子科技有限公司
编辑：《嵌入式技术与智能系统》编委会

网址：<https://www.hanspub.org/journal/etis>
电子邮箱：etis@hanspub.org