基于 MSP432 的 AliOS Things 操作系统移植

杨淇善，韩德强
（北京工业大学 信息学部，北京 100124）

摘　要： 物联网中含有大量的低成本、低功耗并具有组网功能的嵌入式设备。物联网操作系统可比传统操作系统更好地适配此类设备。AliOS Things 作为近年来新型的物联网操作系统，集成了多种面向物联网领域的组件和功能。采用 Cortex-M 内核的 MSP432 系列微控制器具有高性能和超低功耗的优点。以 MSP432P401R 微控制器为例的 AliOS Things 操作系统移植方法与过程，可广泛应用于物联网的终端设备，具有一定的参考价值。

关键词： 物联网操作系统；移植；AliOS Things；MSP432
中图分类号： TP316　　**文献标识码：** A　　DOI：10.16157/j.issn.0258-7998.211924

中文引用格式： 杨淇善，韩德强. 基于 MSP432 的 AliOS Things 操作系统移植[J].电子技术应用，2021，47(11)：95-99.
英文引用格式： Yang Qishan，Han Deqiang. Porting of Alios Things operating system based on MSP432[J]. Application of Electronic Technique，2021，47(11)：95-99.

Porting of Alios Things operating system based on MSP432

Yang Qishan，Han Deqiang
(Faculty of Information Technology，Beijing University of Technology，Beijing 100124，China)

Abstract： The Internet of Things(IoT) contains a large number of embedded devices with low cost，low power consumption and networking functions. The Internet of Things operating system (IoT OS) can better adapt to such devices than the traditional operating system. As a new IoT OS in recent years，Alios Things integrates a variety of components and functions in the field of IoT. MSP432 series microcontrollers with Cortex-M core have the advantages of high performance and ultra-low power consumption. Taking MSP432P401R microcontroller as an example，the porting method and process of Alios Things operating system can shed light on other terminal devices of IoT which need porting operating system.

Key words： Internet of Things operating system；porting；AliOS Things；MSP432

0 引言

物联网(Internet of Things，IoT)是一个通过信息技术将各种物体与网络相连，以帮助人们获取所需物体相关信息的巨大网络[1]。在这个网络中，存在数量巨大的各类嵌入式设备。这些设备在计算能力、可用内存、实时性、通信、功耗等诸多方面都有限制或要求[2]。通用操作系统对硬件系统的性能要求较高，物联网系统内的绝大多数嵌入式设备无法运行。传统的嵌入式操作系统具有内核小、实时性高等优点，适用于性能较低的嵌入式设备。但是，仅有一个轻量级内核并无法解决物联网应用中繁多的通信协议、硬件接口等造成的“碎片化”问题。因此，致力于解决“碎片化”问题、并具有终端安全保障和统一管理技术支撑的物联网操作系统应运而生。

AliOS Things 是面向物联网领域的轻量级物联网操作系统，包含物联网领域需用的组件和接口，具备极致性能、极简开发、云端一体、安全防护等特点，可广泛应用于智能家居、智慧城市、新出行等物联网领域[3]。

TI 公司的 MSP432 系列微控制器采用了 ARM Cortex-M4F 内核，具有浮点运算单元(Float Point Unit，FPU)和存储器保护单元。MSP432P401R 作为该系列的其中一款产品，内置 256 KB 的 Flash 和 64 KB 的 SRAM，最大工作频率 48 MHz，工作功耗和待机功耗分别仅为 80 μA/MHz 和 660 nA。此外，还集成了多个 SPI、UART 和 I^2C 接口，以及密码编译加速器等模块[4]。因此，高性能的特点可充分发挥 AliOS Things 操作系统的优势，超低功耗的特点使其可广泛应用于物联网感知层的设备上。两者的结合在物联网领域中具有良好的应用价值。

1 AliOS Things 系统架构

AliOS Things 物联网操作系统除了提供轻量级的内核以外，还具有网络通信与组网、安全加密、物联网中间件等多种功能与服务组件，其层次架构和组件结构如图1所示。

1.1 BSP

BSP(Board Support Package，板级支持包)是介于硬件

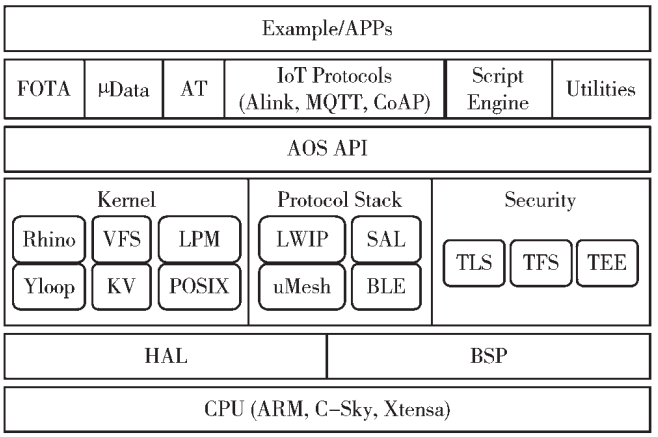

图 1 AliOS Things 操作系统架构图

平台与操作系统之间的一层，封装了访问和使用硬件资源的功能函数。对于微控制器而言，可参考芯片数据手册开发或使用厂商提供的函数库实现。对于硬件平台中的其他外设，则根据实际使用的通信接口和控制方式进行函数封装。

1.2 HAL

HAL(Hardware Abstraction Layer，硬件适配层)屏蔽了硬件平台，使其上层代码不直接访问硬件资源且在不改变功能的情况下可移植到其他平台中，增强了代码的稳定性与移植性。实现该层的函数可根据实际的功能需求调用 BSP 中的函数。

1.3 Kernel

内核层是 AliOS Things 的核心部分，包含 Rhino 实时操作系统内核以及异步事件框架 Yloop、VFS(Virtual File System，虚拟文件系统)等功能组件。其中，Rhino 内核是整个操作系统的基础部分，提供任务调度、内存分配、时间管理和中断控制等一系列功能。移植工作的基本任务即实现操作系统内核在目标平台上正常运行。

1.4 Protocol Stack

协议栈提供了物联网设备连接网络或组网的多种协议。对于面向 IP 的设备，AliOS Things 集成了基于 LwIP 的 TCP/IP 协议栈以及提供标准 Socket 功能的 SAL(Socket Abstraction Layer，套接字抽象层)。对于非 IP 设备，集成了 BLE(Bluetooth Low Energy，低功耗蓝牙)协议和 LoRa 协议。同时，协议栈中还包含可支持不同物联网设备自组织异构网络的 uMesh 技术。

1.5 Security

安全组件中集成了 TLS(Transport Layer Security，安全传输层协议)和 TEE(Trusted Execution Environment，可信执行环境)，并支持 ID2(Internet Device ID，联网设备身份标识)认证与相关密钥管理的功能。

1.6 AOS API

该部分可称为操作系统应用程序编程接口层，针对系统内核以及协议组件进行了封装，将应用层与内核层进行了隔离。上层应用使用操作系统时，仅需调用 AOS API 的接口函数即可，内核代码对其完全透明。因此，只要不改变 AOS API 的接口，修改内核代码或更换协议组件均不影响应用层，提高了系统的可迭代性和维护性。

1.7 中间件

AliOS Things 具有诸多物联网系统可用的中间组件，更好地为整个应用系统提供可靠、便捷的服务。主要包括 FOTA(Firmware Over-The-Air，无线固件升级)、uData 框架、AT 命令解析和常用的物联网云端连接协议 MQTT(Message Queuing Telemetry Transport，消息队列遥测传输)、CoAP(Constrained Application Protocol，受限型应用协议)等。

2 AliOS Things 移植方法和过程

本文的 AliOS Things 移植工作主要涉及系统架构中的以下六个部分：与微控制器架构相关的接口文件、板级支持包、硬件适配层、内核层、AOS API 和 SysTick 定时器中断处理函数。

2.1 接口文件

AliOS Things 的源代码中提供了多种主流微控制器体系架构相关的移植接口文件，以供开发者参考或使用。适用于 MSP432P401R 的 Cortex-M4F 内核和 IAR for ARM 开发环境的相关文件存储于源代码目录 platform\arch\arm\armv7m\iccarm\m4 中，其包含 4 个文件：k_types.h、port.h、port_c.c 和 port_s.S。

k_types.h 对堆栈溢出控制的幻数值、强制内联关键字进行了宏定义。另外，根据微控制器内核的位数，重新定义操作系统中所需用的堆栈、定时器、信号量、互斥量和上下文切换量等数据类型。重新定义上述数据类型时，应使用 C99 标准的头文件 stdint.h 中所定义的数据类型，避免直接使用 C 语言中 char、int 等基本数据类型，以提高代码的可移植性。以 sem_count_t 和 ctx_switch_t 为例：

```
typedef uint32_t sem_count_t;
typedef uint64_t ctx_switch_t;
```

port.h 对任务处理中有关寄存器操作的函数进行了宏定义或声明，这些函数均在 port_c.c 和 port_s.S 两个文件中实现。

port_c.c 使用 C 语言实现了任务堆栈初始化函数 cpu_task_stack_init()。该函数的编写应遵循微控制器内核的相关特性。Cortex-M4F 内核的堆栈增长方式为高地址向低地址递减。此外，当系统产生中断时，寄存器 R0-R3、R12、LR(链接寄存器)、PSR(程序状态寄存器)、S0-S15 和 FPSCR(浮点数状态控制寄存器)均由硬件控制保存与恢复，而寄存器 R4-R11 以及 S16-S31 则需要由软件控制保存与恢复。因此，函数 cpu_task_stack_init()在完成栈指针变量 stk 的初始化之后，可按照以下顺序完成整个任务堆栈的构建。

(1)PSR 的值置为 0x01000000，使能 Thumb 指令集。

```
*(--stk) = (cpu_stack_t)0x01000000L;
```

(2)PC(程序计数器)的值置为任务函数的入口地址。

```
*(--stk) = (cpu_stack_t)entry;
```

(3)LR 的值置为 krhino_task_deathbed()函数指针,该函数用于删除当前意外跳出的任务。

```
*(--stk) = (cpu_stack_t)krhino_task_deathbed;
```

(4)依次入栈 R12、R3-R1 的初值,其数值可根据寄存器编号确定,以便于检查堆栈初始化是否正确。

```
*(--stk) = (cpu_stack_t)0x12121212L;
*(--stk) = (cpu_stack_t)0x03030303L;
*(--stk) = (cpu_stack_t)0x02020202L;
*(--stk) = (cpu_stack_t)0x01010101L;
```

(5)R0 的值置为任务函数传入参数。

```
*(--stk) = (cpu_stack_t)arg;
```

(6)若使用 FPU,则依次入栈寄存器 S31-S16 的初值,其数值与上述通用寄存器类似。

```
*(--stk) = (cpu_stack_t)0x31uL;
*(--stk) = (cpu_stack_t)0x30uL;
……
*(--stk) = (cpu_stack_t)0x16uL;
```

(7)EXC_RETURN 的值置为 0xFFFFFFFD。

```
*(--stk) = (cpu_stack_t)0xFFFFFFFDL;
```

(8)依次入栈 R11-R4 的初值,其数值与上述通用寄存器一致。

```
*(--stk) = (cpu_stack_t)0x11111111L;
*(--stk) = (cpu_stack_t)0x10101010L;
……
*(--stk) = (cpu_stack_t)0x04040404L;
```

除任务堆栈初始化函数以外,port.h 中声明的其他函数使用汇编语言在 port_s.S 中实现。cpu_intrpt_save()和 cpu_intrpt_restore()负责 CPSR(程序状态寄存器)状态的保存和恢复,以及中断的禁止和使能。这两个函数在临界区代码前、后被调用。

```
cpu_intrpt_save:
    MRS     R0, PRIMASK
    CPSID   I
    BX      LR
cpu_intrpt_restore:
    MSR     PRIMASK, R0
    BX      LR
```

另外,还应实现涉及上下文切换功能的部分函数。在任务切换时,需调用 cpu_task_switch()。在中断结束时,需调用 cpu_intrpt_switch()。任务切换和中断结束并非同一种情况,但两者均可利用触发 PendSV 异常处理的方式完成相应的功能。因此,可采用相同的方法处理。

```
LDR     R0, =SCB_ICSR
LDR     R1, =ICSR_PENDSVSET
STR     R1, [R0]
BX      LR
```

cpu_first_task_start()则用于多任务系统的初始化阶段,由优先级最高的就绪任务开始执行。首先,该函数将 PendSV 优先级设为 14 级(PRI_14),Systick 优先级设为 15 级(PRI_15)。

```
LDR     R0, =SHPR3_PRI_14
LDR     R1, =PRI_LVL_PENDSV
STRB    R1, [R0]
LDR     R0, =SHPR3_PRI_15
LDR     R1, =PRI_LVL_SYSTICK
STRB    R1, [R0]
```

随后,PSP(进程堆栈指针)赋值为 0,以 8 字节对齐 MSP(主堆栈指针)。

```
MOVS    R0, #0
MSR     PSP, R0
MRS     R0, MSP
LSRS    R0, R0, #3
LSLS    R0, R0, #3
MSR     MSP, R0
```

最后,触发 PendSV 异常并开启中断。

```
LDR     R0, =SCB_ICSR
LDR     R1, =ICSR_PENDSVSET
STR     R1, [R0]
CPSIE   I
B       .
```

PendSV 异常处理函数 PendSV_Handler()也在 port_s.S 中实现。通过 PSP 是否为 0 可判断触发该异常处理的函数。

```
MRS     R0, PSP
CBZ     R0, _pendsv_handler_nosave
```

若由 cpu_task_switch()和 cpu_intrpt_switch()触发,则保存通用寄存器的值,并将状态寄存器赋值给对应的任务模块。反之,若由 cpu_first_task_start()触发异常,则跳过上述过程,直接从代码段 _pendsv_handler_nosave 继续执行。然后,比较任务的优先级并执行当前最高级就绪的任务。

2.2 板级支持包

对于移植工作,BSP 需要对时钟、中断、SysTick、GPIO 等功能实现支持。MSP432P401R 片内 32 KB ROM 内集成了 TI 官方的 MSP432 外设驱动程序库,仅需将相关头文件加入相关源文件中即可。库函数均以 ROM_ 作为前缀,以全局中断使能函数为例:

```
ROM_Interrupt_enableMaster();
```

2.3 硬件适配层

AliOS Things 的源代码中提供了 HAL 函数声明的参考,相关文件存储于源代码目录 include\hal\soc 中。移植工作根据实际情况,重新声明并实现 HAL 的函数。相关函数均以 HAL_ 作为前缀,以 SysTick 初始化和系统初始化两个函数为例:

```
void  HAL_SysTick_Init(void)
```

```
{
    ROM_SysTick_enableModule();
    ROM_SysTick_setPeriod(SystemCoreClock
            /RHINO_CONFIG_TICKS_PER_SECOND);
}
void HAL_System_Init(void)
{
    HAL_SysTick_Init();
    HAL_LED_Init();
    HAL_SysTick_Enable_Interrupt();
    ROM_Interrupt_enableMaster();
}
```

2.4 内核层

开发者不应更改 Rhino 内核的源代码，即以 krhino_ 为前缀的函数，避免出现不可预知的错误。但是，需要根据实际硬件平台和系统需求，利用诸多宏定义配置和剪裁操作系统内核功能。AliOS Things 源代码中的头文件 k_default_config.h 给出了用于配置和剪裁内核功能的相关宏定义的默认值，该文件存储于源代码目录 kernel\rhino\core\include 中。该文件中对微控制器内核相关的特性进行了配置，默认采用小端模式，由高地址向低地址增长的堆栈方向等。

```
#define    RHINO_CONFIG_LITTLE_ENDIAN      1
#define    RHINO_CONFIG_CPU_STACK_DOWN     1
```

除此之外，该文件中的几十个宏定义还配置了操作系统的时标频率、信号量、消息队列、互斥量等功能特性，以及动态内存分配、中断堆栈检查、钩子函数等系统功能的默认值。

若修改这些宏定义的值，应在同目录下创建一个名为 k_config.h 的头文件，而非直接修改 k_default_config.h，以保持代码的可移植性和正确性。由于 k_default_config.h 中采用了 #ifndef…#define…预编译，则 k_config.h 中仅对要修改的默认值重新定义即可。

2.5 AOS API

AliOS Things 源代码中提供了 AOS API 函数声明的参考，该层函数以 aos_ 作为前缀，相关头文件存储于源代码目录 include\aos 中。开发者不必逐一实现所有头文件中声明的函数，根据系统情况实现所需函数即可。本文的移植工作主要利用 Rhino 内核相应的函数完成本层函数的实现，以 aos_init()和 aos_task_new_ext()两个函数为例：

```
void aos_init(void)
{
    krhino_init();
}
int aos_task_new_ext(aos_task_t *task, const char *name,
    void (*fn)(void *), void *arg, int stack_size, int prio)
{
    int ret;
    ret=(int)krhino_task_dyn_create((ktask_t**)(&(task->
    hdl)),name, arg, prio, 0, stack_size / sizeof(cpu_
    stack_t), fn, 1u);
    if (ret == RHINO_SUCCESS) { return 0; }
    ERRNO_MAPPING(ret);
}
```

2.6 SysTick 定时器中断处理函数

SysTick 定时器为操作系统提供了“心跳”功能，即操作系统调度所需要的最小基本定时单元。根据不同的任务调度策略，当 SysTick 中断触发时，操作系统会采取相应的措施进行多任务的切换。Rhino 内核中提供了平台无关的入口函数 krhino_tick_proc()，以供 SysTick 中断处理函数调用。在执行过程中，krhino_tick_proc()应受临界段代码保护。

```
void SysTick_Handler(void)
{
    krhino_intrpt_enter();
    krhino_tick_proc();
    krhino_intrpt_exit();
}
```

3 测试

移植工作基于 MSP-EXP432P401R LaunchPad 开发平台实现，其具有两个 LED 可供测试工作使用，提供直观的视觉指示。同时，该平台自带 XDS110 板载调试器，可通过 USB 接口直接与上位机相连，实现程序的下载和调试功能。软件开发环境采用了 IAR Embedded Workbench for ARM 8.22，通过配置相关参数即可支持 MSP432P401R 和 XDS110 的开发和调试，如图 2 所示。

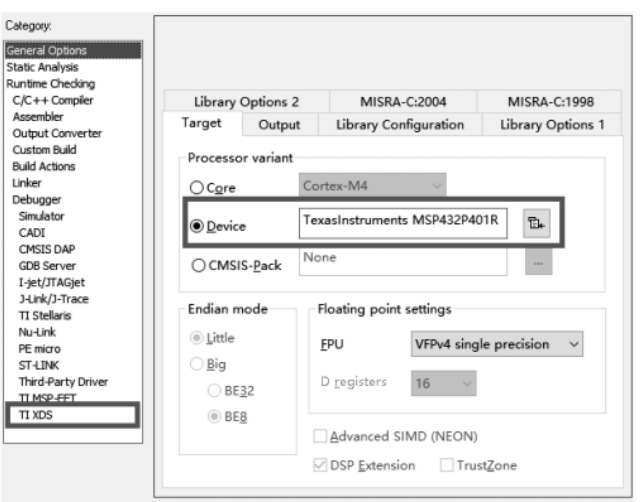

图 2 IAR 开发环境配置

测试程序中，通过查看多个任务的运行结果，以确定操作系统移植后能否正常运行。首先，调用 HAL 和 AOS API 的初始化函数，分别对硬件资源和操作系统内核进行初始化。然后，创建两个测试任务 LED1_Blink_Task

和 LED2_Blink_Task。其中，LED2_Blink_Task 的优先级高于 LED1_Blink_Task。最后，调用 aos_start()函数启动操作系统内核，使系统进入运行状态。主函数流程图如图 3 所示。

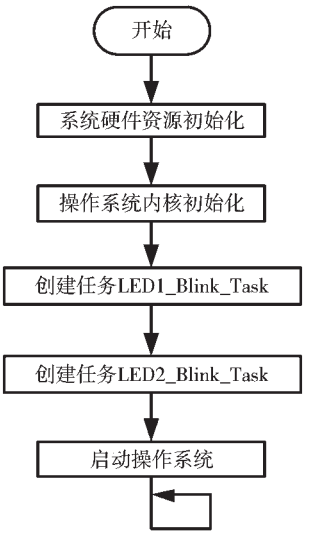

图 3　测试程序主函数流程图

两个测试任务分别使开发平台的 LED1 和 LED2 以 300 ms 和 2 000 ms 的周期翻转。以 LED1_Blink_Task 为例：

```
HAL_LED1_Toggle();
aos_msleep(300);
```

此外，测试程序中设置了全局数组 gSysTime[]，通过调用 aos_now_ms()函数在该数组中记录每次任务开始执行的时间点。在 LED1_Blink_Task 中，gSysTime[]直接记录 aos_now_ms()函数返回的时间值。

```
gSysTime[gArrayCount++]=aos_now_ms();
```

而在 LED2_Blink_Task 中，将 aos_now_ms()函数的返回值加 1 后存入 gSysTime[]中。

```
gSysTime[gArrayCount++]=aos_now_ms()+1;
```

由于两个任务的执行周期均为 100 ms 的整数倍，则某个时间点上两个任务同时就绪并相继执行时，以此可区分数组中时间值的来源。

测试程序运行后，可看到开发平台的 LED1 和 LED2 各自按照一定的周期闪烁。同时，通过 IAR 开发环境中的 Watch 窗口，查看测试程序运行一段时间后数组 gSysTime[]的状态，结果如图 4 所示。

由 gSysTime[0]=1 和 gSysTime[1]=0 可知，系统启动后，即系统时间 0 ms 时，两个任务均处于就绪状态并先执行了优先级更高的 LED2_Blink_Task。此后的 2 000 ms 内，gSysTime[2]至 gSysTime[7]的数值表明 LED1_Blink_Task 以 300 ms 的周期正确执行。图 4 中的 gSysTime[8]、gSysTime[16]和 gSysTime[23] 的数值表明 LED2_Blink_Task 以 2 000 ms 的周期正确执行。此外，在系统时间6 000 ms 时，两个任务再次同时就绪，gSysTime[23]=6001 和 gSysTime[24]=6000 表明操作系统以优先级执行了两个任务。

Watch 1

Expression	Value	Location
gSysTime	<array>	0x20000BD8
[0]	1	0x20000BD8
[1]	0	0x20000BE0
[2]	300	0x20000BE8
[3]	600	0x20000BF0
[4]	900	0x20000BF8
[5]	1200	0x20000C00
[6]	1500	0x20000C08
[7]	1800	0x20000C10
[8]	2001	0x20000C18
[9]	2100	0x20000C20
[10]	2400	0x20000C28
[11]	2700	0x20000C30
[12]	3000	0x20000C38
[13]	3300	0x20000C40
[14]	3600	0x20000C48
[15]	3900	0x20000C50
[16]	4001	0x20000C58
[17]	4200	0x20000C60
[18]	4500	0x20000C68
[19]	4800	0x20000C70
[20]	5100	0x20000C78
[21]	5400	0x20000C80
[22]	5700	0x20000C88
[23]	6001	0x20000C90
[24]	6000	0x20000C98
[25]	6300	0x20000CA0

图 4　数组 gSysTime 的测试结果

综上所述，完成 AliOS Things 操作系统移植后，多个任务能够正确地创建、运行和切换。

4　结论

物联网操作系统起源于 TinyOS 和 Contiki 项目，它们对 IoT 系统的发展产生了深远的影响[5]。本文给出了 AliOS Things 操作系统移植至 MSP432 微控制器的方法，以操作系统架构中由底层至顶层的顺序详细描述了各层中需要完成的工作。通过相关测试，AliOS Things 能够正常、稳定地运行在 MSP432 微控制器上，验证了移植方法的正确性，对于其他平台具有借鉴意义和参考价值。

参考文献

[1] 桂小林，张学军，赵建强，等.物联网信息安全[M].北京：机械工业出版社，2014.
[2] 郝继锋，李运喜，徐晓光.开源低端设备物联网操作系统研究[J].信息通信，2018(4)：155-157.
[3] 史治国，陈积明.物联网操作系统 AliOS Things 探索与实践[M].杭州：浙江大学出版社，2018.
[4] Texas Instruments.MSP432P401R，MSP432P401M SimpleLink Mixed-Signal Microcontrollers[Z].2019.
[5] 何小庆.物联网操作系统展望[J].单片机与嵌入式系统应用，2021，21(2)：6.

(收稿日期：2021-07-09)

作者简介：
杨淇善(1988-)，男，硕士研究生，实验师，主要研究方向：嵌入式系统设计及应用。
韩德强(1966-)，男，硕士，高级实验师，主要研究方向：嵌入式系统教学与应用。

扫码下载电子文档

《电子技术应用》http://www.chinaaet.com

版权声明