Embedded Technology and Intelligent Systems 嵌入式技术与智能系统, 2025, 2(2), 59-77
Published Online April 2025 in Hans. https://www.hanspub.org/journal/etis
https://doi.org/10.12677/etis.2025.22006

Hans汉斯

基于Zephyr RTOS的嵌入式软件开发实践

何灵渊[1]，何小庆[2]

[1]博通公司，美国 森尼韦尔
[2]嵌入式系统联谊会，北京

收稿日期：2025年4月1日；录用日期：2025年4月23日；发布日期：2025年4月30日

摘　要

Zephyr开源项目由Linux基金会维护，是一个针对资源受限的嵌入式设备优化的小型、可缩放、多体系结构实时操作系统(RTOS)。近年来，Zephyr RTOS在嵌入式开发中的采用度逐步增加，支持的开发板和传感器不断增加，其广泛的设备支持和高度的可扩展性吸引了开发者的关注。相比FreeRTOS等小型RTOS而言，教育生态不够成熟的Zephyr系统规模更大，结构更复杂，这提高了开发者入门和精通的门槛。文章对Zephyr硬件抽象层和设备驱动的架构与实现进行系统性分析，重点阐述了设备驱动模型和设备树的作用。为了展示基于Zephyr的嵌入式软件开发，文章在BBC micro:bit V2开源硬件上构建样例Zephyr设备驱动和应用程序，并做解释和验证。

关键词

嵌入式系统，软件开发，实时操作系统，Zephyr项目

Embedded Software Development Practice with Zephyr RTOS

Lingyuan He[1], Xiaoqing He[2]

[1]Broadcom Inc., Sunnyvale, USA
[2]Embedded System Association, Beijing

Received: Apr. 1st, 2025; accepted: Apr. 23rd, 2025; published: Apr. 30th, 2025

Abstract

Zephyr, an open-source initiative managed by the Linux Foundation, is a small, scalable, multi architecture Real-time Operating System (RTOS) optimized for resource-constrained embedded systems. In recent years, the adoption rate of Zephyr RTOS has increased significantly. The range of supported

文章引用: 何灵渊, 何小庆. 基于 Zephyr RTOS 的嵌入式软件开发实践[J]. 嵌入式技术与智能系统, 2025, 2(2): 59-77.
DOI: 10.12677/etis.2025.22006

boards and sensors continues to rise. Developers are increasingly interested in Zephyr because of its board device support and scalability. Compared to a minimum RTOS like FreeRTOS, Zephyr, whose education infrastructure has not yet matured, has a larger scale and a more complex architecture. This means Zephyr's learning curve for a developer is steep. This paper systematically describes the architecture and implementation of Zephyr's hardware abstraction and device driver, especially its device driver model and its usage of the device tree. It also demonstrates embedded software development with Zephyr by building, analyzing, and verifying a custom driver and an example application based on the BBC micro:bit V2 open-source hardware.

Keywords

Embedded System, Software Development, Real-Time Operating System, Zephyr Project

1. 引言

Zephyr 是由 Linux 基金会管理的开源实时操作系统(RTOS) [1]，其前身为用于数字信号处理的 Virtuoso 操作系统，后被风河(Wind River)收购，更名为 Rocket RTOS。2016 年它成为了 Linux 基金会的项目，更名为 Zephyr。

Zephyr 得到了多家半导体企业的支持，包括恩智浦、意法半导体、瑞萨、北欧半导体(Nordic)、英特尔和德州仪器等，并已经被应用到了众多设备中，覆盖了消费电子、能源、医疗、工业、农业等领域[2]。Zephyr 的 Apache 2.0 开源协议授权让它在非商用和商用解决方案中都可免费使用[3]。

近年来 Zephyr 的热度逐渐上升，在嵌入式开发中的采用度逐步增加。Eclipse 基金会的《2024 年物联网和嵌入式开发者调查报告》表明，在资源受限设备上使用 Zephyr 的开发者从 2022 年的 8%增长到了 2024 年的 21%，这已经和裸机直接编程的比例相当，也非常接近第二位的 FreeRTOS (29%) [4]。

相比 FreeRTOS 等小型 RTOS 而言，教育生态不够成熟的 Zephyr 系统规模更大，结构更复杂，这提高了开发者入门和精通的门槛。本文对 Zephyr 硬件抽象层和设备驱动的架构与实现进行系统性分析，重点阐述了设备驱动模型和设备树的作用。为了展示基于 Zephyr 的嵌入式软件开发，本文在 BBC micro:bit V2 开源硬件上构建样例 Zephyr 设备驱动和应用程序，并做解释和验证。

2. Zephyr 的硬件抽象层和配置概述

Zephyr 有着完善的设备驱动支持，而且高度可配置。作为 Linux 基金会的项目，它用到了和 Linux 内核类似的工具，特别是设备树(Device Tree)和 Kconfig 配置语言。本章将对与开发息息相关的硬件抽象化和配置进行概述。

2.1. 设备驱动模型

Zephyr 的设备驱动模型负责初始化系统中所有的驱动程序，为系统中的所有设备驱动提供了统一的配置方法[5]。如图 1 所示的是设备驱动模型的概览。

Zephyr 中每一种子系统驱动(UART、I2C 等)都有着泛用类型(Generic Type，非设备特定)的接口，具体的驱动实现会提供实现这些驱动接口函数的指针。在图 1 中可以看到，在子系统 2 中有两种设备驱动

的实例，但是两种驱动都会提供泛用 API 1 到 3 的实现。应用程序代码可以在兼容的设备上直接使用泛用 API，具体驱动的实现代码会被调用。如子系统 1 中所示，同一种驱动可以在系统中多次实例化，比如多个 UART 接口。

设备驱动代码在初始化时也会为每个设备提供驱动特定的配置，即图 1 中的 struct config。在实际代码中这可能是通过 Kconfig 配置的参数，比如显示器的刷新频率。驱动代码还可以为每个驱动指定一个结构用于存储相关的数据。

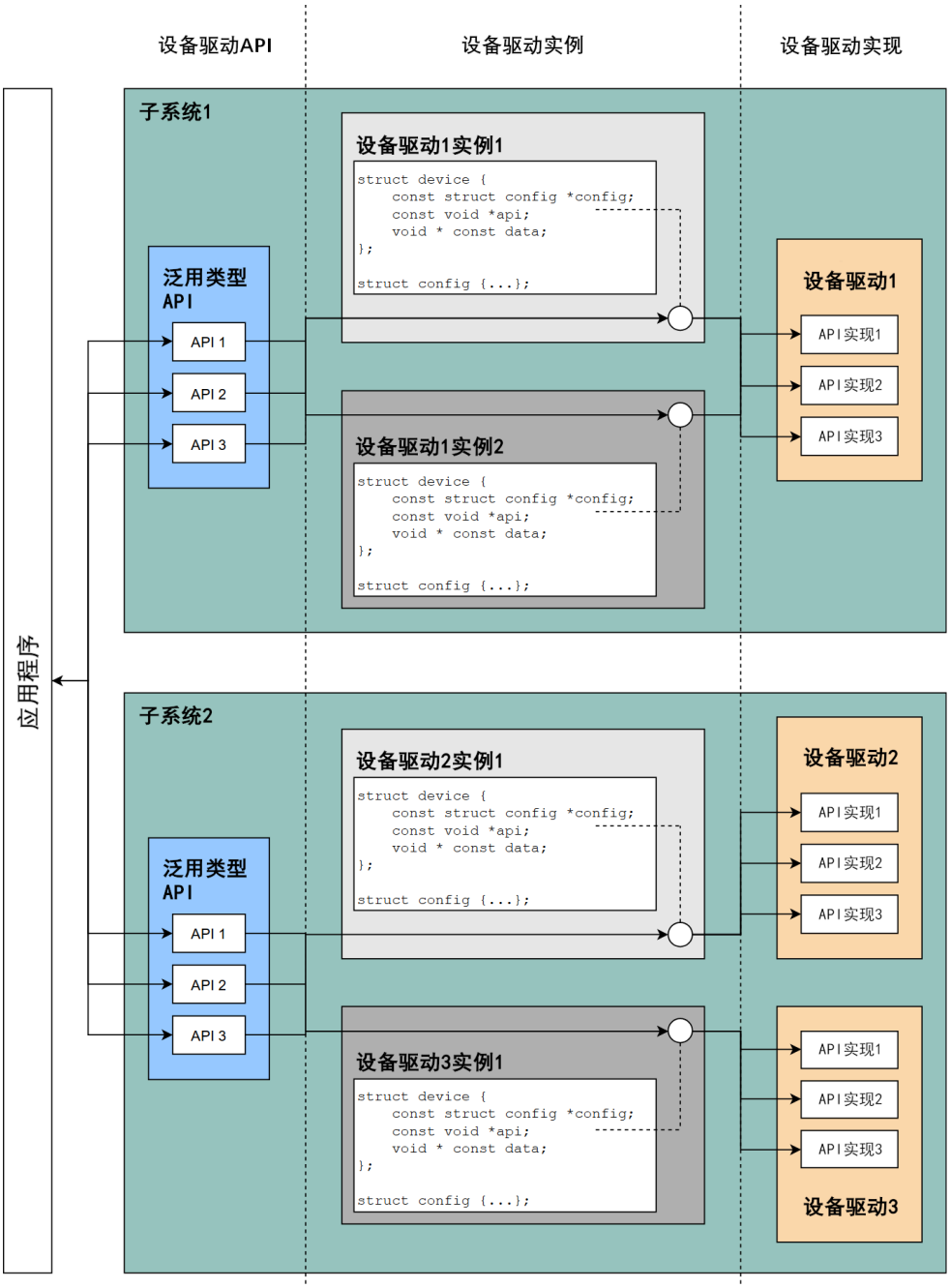

Figure 1. An overview of the device driver model (source: zephyrproject.org)
图 1. 设备驱动模型概览(来源：zephyrproject.org)

一个驱动的泛用接口定义会出现在驱动的头文件中，图 2 中定义了 subsystem 子系统的泛用接口 subsystem_do_this 和 subsystem_do_that 函数。图 3 中的 my_driver 驱动实现了自己的 do_this 和 do_that 函数，并将它们的指针填入了驱动 API 结构(do_this 和 do_that 成员)。注意应用程序代码应该直接使用 subsystem_do_this/that 函数，这两个函数会通过 DEVICE_API_GET 宏进入正确的驱动接口实现，即 my_driver_do_this/that 函数。在实际的驱动中，subsystem 会被替代为能够代表设备的名称，例如在通用的显示驱动接口(include/zephyr/drivers/display.h)中，subsystem 被替代为了 display。

```
typedef int (*subsystem_do_this_t)(const struct device *dev, int foo, int bar);
typedef void (*subsystem_do_that_t)(const struct device *dev, void *baz);

__subsystem struct subsystem_driver_api {
      subsystem_do_this_t do_this;
      subsystem_do_that_t do_that;
};

static inline int subsystem_do_this(const struct device *dev, int foo, int bar)
{
      return DEVICE_API_GET(subsystem, dev)->do_this(dev, foo, bar);
}

static inline void subsystem_do_that(const struct device *dev, void *baz)
{
      DEVICE_API_GET(subsystem, dev)->do_that(dev, baz);
}
```

Figure 2. A sample driver interface definition (source: zephyrproject.org)
图 2. 样例驱动接口定义(来源：zephyrproject.org)

```
static int my_driver_do_this(const struct device *dev, int foo, int bar)
{
      ...
}

static void my_driver_do_that(const struct device *dev, void *baz)
{
      ...
}

static DEVICE_API(subsystem, my_driver_api_funcs) = {
      .do_this = my_driver_do_this,
      .do_that = my_driver_do_that,
};
```

Figure 3. A sample driver implementation (source: zephyrproject.org)
图 3. 样例驱动实现(来源：zephyrproject.org)

在进行具体子系统驱动的实例化时，驱动代码还会提供初始化代码和初始化的优先级。

2.2. 设备树

设备树(Device Tree)是用于描述硬件的层级化数据结构。设备树规范[6]描述了设备树的概念、用途、结构、设备树绑定(binding)和设备树语言。

2.2.1. 设备树的作用

Zephyr 和 Linux 同样使用设备树，Zephyr 为了减少运行时的数据和代码，会使用设备树的数据产生 C 语言头文件[7]。Zephyr 中定义了一整套宏，用于访问设备树节点和取得设备树节点的属性。

Zephyr 中设备树有两项主要作用：

- 在设备驱动模型中描述硬件。
- 提供硬件的初始配置。

设备树和 Kconfig 在 Zephyr 中都起到了配置语言的作用，设备树用于描述硬件和启动时的配置，Kconfig 则主要用于配置软件。

设备树有两种输入文件：设备树源文件和设备树绑定[8]。源文件描述了设备树本身，绑定则用于描述设备树的内容，特别是数据类型和结构。Zephyr 在构建时使用这两种文件生成 C 头文件，devicetree.h 头文件提供通用的宏访问设备树(以“DT_”打头)。

2.2.2. 设备树的语法

图 4 所示的是一个最小的样例设备树源文件[9]：

```
/dts-v1/;

/ {
        a-node {
                subnode_nodelabel: a-sub-node {
                        foo = <3>;
                };
        };
};
```

Figure 4. A minimum device tree file (source: zephyrproject.org)
图 4. 设备树最小样例(来源：zephyrproject.org)

图中“/”代表根节点，a-node 是根节点的子节点，a-sub-node 是 a-node 的子节点，a-sub-node 还有一个 label (标签) subnode_nodelabel。标签是可选的，在设备树中每个标签只能出现一次，代码可以通过标签直接访问节点。每个节点都有自己的路径，和 Linux 文件路径相似，例如 a-sub-node 的全路径为：/a-node/a-sub-node。

图 5 所示的是一个较为贴近实际硬件的设备树样例：

```
/dts-v1/;

/ {
        soc {
                i2c@40003000 {
                        compatible = "nordic,nrf-twim";
                        reg = <0x40003000 0x1000>;

                        apds9960@39 {
                                compatible = "avago,apds9960";
                                reg = <0x39>;
                        };
                        ti_hdc@43 {
                                compatible = "ti,hdc", "ti,hdc1010";
                                reg = <0x43>;
                        };
                        mma8652fc@1d {
                                compatible = "nxp,fxos8700", "nxp,mma8652fc";
                                reg = <0x1d>;
                        };
                };
        };
};
```

Figure 5. A complete device tree example (source: zephyrproject.org)
图 5. 一个完整的设备树样例(来源：zephyrproject.org)

在图 5 中可以看到节点名的命名方法为“总线类型或设备名@地址”，这样的惯例不仅有助于区分类似的节点，还能够帮助快速确定节点指向的设备和总线类型。地址的惯例根据设备类型有所不同：

- 在内存中映射的外设：使用寄存器映射的基地址，例如 i2c@40003000 表示 I2C 映射的寄存器基地址为 0x40003000。
- I2C 外设：使用外设在 I2C 总线上的地址，例如 apds9960 的 I2C 地址为 0x39。
- SPI 外设：使用外设的片选线序号，如果没有则使用 0。
- 内存：使用物理内存的起始地址，例如 memory@2000000 表示从 0x2000000 物理地址开始的 RAM。
- 在内存中映射的闪存：和 RAM 类似使用物理起始地址，例如 flash@8000000。
- 固定的闪存分区：使用分区的偏移量，例如在 flash@8000000 设备中可以有一个 partitions 节点代表分区表，其中有 partition@0 和 partition@20000 两个节点，分别意味着起始地址 0x8000000 和 0x8020000 的两个分区。

设备树节点中每个属性有一个名称和一个值，属性的值可以是字符串、整型数、布尔值、8 位整型数组、字符串数组、混合类型数组、指向节点的 phandle (类似 C 语言中的指针)、复数的 phandle 或是 phandle 数组。

设备树节点中几个重要的属性如下：

- compatible：表示节点所代表的硬件设备，本文翻译为兼容名。兼容名属性在构建过程中十分重要，驱动程序通过兼容名的值查找可以适配的硬件。兼容名的值可以是字符串数组，将数个驱动程序从最特定到最泛用进行排列，首个匹配的驱动程序会被加载。
- reg：用于设备寻址，其格式为 16 进制的<地址，长度>。
- status：用于表示节点是否启用。Zephyr 支持“okay”和“disabled”，分别表示启用和禁用。节点必须启用，Zephyr 的驱动模型才会应用到节点上。

除了标签，设备树源文件中还可以定义 chosen (选择)和 alias (别名)来帮助应用代码或驱动寻找特定的节点，如图 6 所示。

```
/dts-v1/;

/ {
    chosen {
            zephyr,console = &uart0;
    };

    aliases {
            my-uart = &uart0;
    };

    soc {
            uart0: serial@12340000 {
                    ...
            };
    };
};
```

Figure 6. Use chosen and aliases nodes in a device tree file (source: zephyrproject.org)
图 6. 在设备树中使用 chosen 和 aliases 节点(来源：zephyrproject.org)

图中/alias 和/chosen 节点都不指向实际的硬件设备，它们被用来指定设备树中的其他节点：my-uart是/soc/serial@12340000 路径的别名(uart0 标签名)，uart0 标签还被选为“zephyr, console”。选择和别名可以帮助抽象化不同的开发板，例如闪灯样例(samples/basic/blinky/src/main.c)中使用 led0 别称节点达到支持多种开发板的目的，只要开发板的设备树文件中有别称为 led0 的节点，样例即可运行。

Zephyr 中每个支持的开发板都有自己的主设备树文件，micro:bit V2 的文件位于路径 boards/bbc/microbit_v2/bbc_microbit_v2.dts，其中可以看到 GPIO 按钮、LED 显示矩阵、I2C 总线和 I2C 总线上的传感器等硬件。应用也可以提供专门针对开发板的设备树覆盖文件，路径为“<应用或模块路径>/boards/<开发板名>.overlay”。覆盖文件中可以增加新的选择/别名节点，也可以配合新的设备树绑定文件(见下节)增加节点。

2.2.3. 设备树绑定

设备树自身的结构相对自由，需要有设备树绑定才能够正确、完整地描述硬件[10]。设备树绑定中包含对设备树节点格式和内容的要求。Zephyr 使用 YAML 文件存储设备树绑定。

图 7 所示的是一个样例绑定文件[11]：

```
description: Custom properties

# The following compatible key does not use the value "custom,props-basics"
# since the vendor "custom" doesn't exist in the vendor-prefixes.txt.
# The generation doesn't fail but it leads to the following message:
# compatible 'custom-props-basics' has unknown vendor prefix 'dummy'
compatible: "custom-props-basics"

properties:
  existent-boolean:
    type: boolean
  int:
    type: int
    required: true
  array:
    type: array
  uint8-array:
    type: uint8-array
  string:
    type: string
  string-array:
    type: string-array
  enum-int:
    type: int
    enum:
      - 100
      - 200
      - 300
  enum-string:
    type: string
    enum:
      - "whatever"
      - "works"
```

Figure 7. A sample device tree binding file (source: Martin Lampacher’s code on GitHub)
图 7. 一个样例设备树绑定文件(来源：Martin Lampacher 在 GitHub 上的代码)

从图 7 中可以看到 3 个重要的键值[12]：

- description (描述)：描述绑定文件适配的硬件的字符串。
- compatible (兼容名)：和设备树中的兼容名对应，一个绑定文件的兼容名如果和一个设备树节点一致，则该设备树节点的格式应当符合绑定文件的内容。
- properties (属性)：描述了符合绑定的节点中的属性与格式。

图 8 所示的是设备树节点符合图 7 中的定义：

```
label_with_props: node_with_props {
  compatible = "custom-props-basics";
  existent-boolean;
  int = <1>;
  array = <0xA second_value: 0xB 0xC>;
  uint8-array = [ 12 34  ];
  string = string_value: "foo bar baz";
  string-array = "foo", "bar", "baz";
  enum-int = <200>;
  enum-string = "whatever";
 };
};
```

Figure 8. A device tree node that is compatible with the binding (source: Martin Lampacher's code on GitHub)
图 8. 符合绑定文件的设备树节点(来源：Martin Lampacher 在 GitHub 上的代码)

从图 8 中可以看到：

- 节点的兼容名和绑定的一致。
- 每一个属性都有按照绑定中 type 的类型赋值。

Zephyr 中默认包括的绑定文件位于 dts/bindings 子目录下，按照类型进行分类，以兼容名的名称进行命名。

除非向 Zephyr 中添加新的硬件支持，一般开发中不添加新的绑定文件。需要时应用可以增加新的绑定文件(<应用或模块路径>/dts/bindings/<兼容名>.yaml)，并在设备树覆盖文件中添加符合绑定定义的节点。

2.2.4. 在程序中访问设备树节点和属性

从 C/C++应用代码中可以用多种方式访问设备树节点。

```
/dts-v1/;

/ {

        aliases {
                sensor-controller = &i2c1;
        };

        soc {
                i2c1: i2c@40002000 {
                        compatible = "vnd,soc-i2c";
                        label = "I2C_1";
                        reg = <0x40002000 0x1000>;
                        status = "okay";
                        clock-frequency = < 100000 >;
                };
        };
};
```

Figure 9. Methods to access a device tree node (source: zephyrproject.org)
图 9. 访问设备树节点的方法(来源：zephyrproject.org)

以图 9 为例，多种宏都可以得到 i2c@40002000 节点(注意：将所有不是字母数字的字符替换为下划线)：

- DT_PATH(soc, i2c_40002000)：将全路径以逗号隔开，省略所有“/”。
- DT_NODELABEL(i2c1)：使用标签名。
- DT_ALIAS(sensor_controller)：使用别名。
- DT_INST(x, vnd_soc_i2c)：寻找第 x 个兼容名为“vnd,soc-i2c”的节点。在本例中因为只有一个节点，x 应为 0。在多“vnd, soc-i2c”节点的情况下，x 和设备树中节点的对应关系不能保证。

对于 chosen 节点(图 9 中不包括)，使用 DT_CHOSEN 指定节点，例如针对图 6 中的设备树可以使用“DT_CHOSEN(zephyr_console)”。

注意：上述宏不能用于变量，只能用于宏定义。

DT_NODE_HAS_PROP 宏可以用于检测节点是否有特定属性，例如“DT_NODE_HAS_PROP(DT_NODELABEL(i2c1), clock_frequency)”的值为 1。访问节点的属性时使用 DT_PROP 宏，例如“DT_PROP(DT_PATH(soc, i2c_40002000), clock_frequency)”的值为 100000。DT_PROP 的值可以用于变量初始化或是静态定义。

Zephyr 定义了众多与设备树相关的宏，在官方文档中有分类总结。在开发中请根据需要查阅文档，并参考 Zephyr 丰富的开发板/传感器样例库。

2.3. Kconfig 配置工具

Kconfig 是在构建时配置 Zephyr 内核和子系统的主要方式，Kconfig 也是 Linux 内核的配置系统。Zephyr 中的 Kconfig 配置选项按照文件夹的层级结构分布，从 Zephyr 代码库根目录的 Kconfig.zephyr 文件开始。根 Kconfig 文件用包含(include)语句包括了子系统(例如内核、驱动和代码库)的 Kconfig 文件，子系统还可以进一步深入定义更深层的 Kconfig 结构和选项。

开发板和应用可以指定需要启用的配置。BBC micro:bit V2 板的默认选项位于文件 boards/bbc/micro-bit_v2/bbc_microbit_v2_defconfig 中，包括系统时钟、串口和控制台等选项。每个应用中的 prj.conf 则包含了应用所需的选项。

与 Linux 类似，在 Zephyr 中可以通过命令行界面进行 Kconfig 选项配置[13]。针对应用构建后产生的 build 文件夹运行命令“west build --build-dir ./build -t menuconfig”即可进入命令行界面(见图 10)。

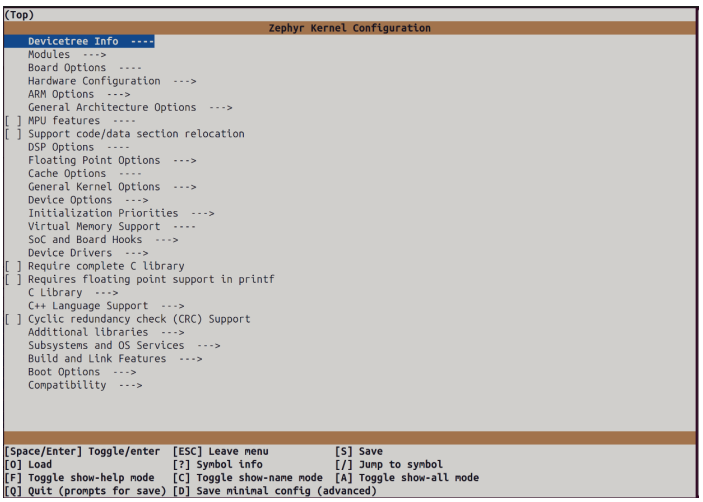

Figure 10. Kconfig menuconfig interface
图 10. Kconfig 配置命令行界面

在界面中可以通过方向键和 ESC/空格键进行导航，在选项上通过空格键进行选择。修改选项后 D 键保存最小配置到文件，也就是当前界面中定义的 Kconfig 选项和 Zephyr 定义的开发板默认选项的区别。图 11 所示的是 micro:bit V2 LED 矩阵显示样例的输出结果(第 3.1 节会使用这一样例)：

```
CONFIG_SERIAL=y
CONFIG_GPIO=y
CONFIG_CONSOLE=y
CONFIG_NRFX_GPIOTE_NUM_OF_EVT_HANDLERS=1
CONFIG_CLOCK_CONTROL_NRF_K32SRC_RC=y
CONFIG_CLOCK_CONTROL_NRF_K32SRC_250PPM=y
CONFIG_UART_CONSOLE=y
CONFIG_DISPLAY=y
CONFIG_MICROBIT_DISPLAY=y
```

Figure 11. Kconfig minimum config output
图 11. Kconfig 最小配置输出

对比前面提到的开发板默认 Kconfig 选项和应用添加的选项(samples/boards/bbc/microbit/display/prj.conf)，可以看到只有“CONFIG_NRFX_GPIOTE_NUM_OF_EVT_HANDLERS”选项是上述两个文件中没有包括的，这是因为北欧半导体的 HAL 层自动定义了这一选项(modules/hal_nordic/nrfx/nrfx_kconfig.h)。除了这样的例外情况，一般在命令行界面中选中了新的选项，用最小选项输出就可以帮助确定新的选项名，之后就可以将其加入到 prj.conf 文件中，从而在编译过程中包括这一选项。

Kconfig 选项除了用于开启子系统功能之外，也用于配置驱动、应用代码，以及下一章将要讲解的日志系统。在代码中可以用“CONFIG_<Kconfig 配置名>”宏取得配置值，构建用的 CMakeLists.txt 文件也可以用“CONFIG_<Kconfig 配置名>”读取配置值。

3. 基于 Zephyr 的嵌入式应用开发

本章中，我们将结合样例在 Zephyr 上实践嵌入式应用开发，帮助理解上一章中的理论。

3.1. 环境配置和运行第一个程序

首先，跟随 Zephyr 项目入门指南(https://docs.zephyrproject.org/latest/develop/getting_started/index.html)完成环境配置、Zephyr 和 Zephyr SDK 的安装。总体来说，Zephyr 在 Linux 中的安装和配置步骤最为简洁，推荐在 Ubuntu Linux 上进行 Zephyr 的实验和开发。本章提及的命令和环境细节均以在 Ubuntu 24.04 版本上使用 Zephyr 4.0.99 开发版本为准，运行时使用 BBC micro:bit V2 开发板(见图 12)。

Figure 12. micro:bit V2 board (source: microbit.org [14])
图 12. micro:bit V2 板(来源：microbit.org [14])

Zephyr 的样例库中包括众多开发板和传感器的样例，不过指南中提到的闪灯样例(Blinky，路径 samples/basic/blinky)并不能直接套用在 micro:bit V2 上。此处我们采用 micro:bit V2 的 LED 矩阵显示样例(路径 samples/boards/bbc/microbit/display)。连接开发板到 Ubuntu 系统上，运行图 13 中的命令进行编译和烧录。命令中的“-p”选项意味着进行全新编译，当对工程进行重复编译时使用“-p auto”选项允许 west 工具只对更改的部分进行重新编译，这适合在开发迭代时节约时间。

```
west build -b bbc_microbit_v2 samples/boards/bbc/microbit/display -p
west flash
```

Figure 13. Commands to compile and flash the sample onto the micro:bit V2 board
图 13. 针对 micro:bit V2 板编译和烧录的命令

成功后开发板会自动启动 Zephyr，开发板背后(见图 12，有 BBC micro:bit v2 字样的面为正面) 5 乘 5 的 LED 矩阵会显示数字倒计时 9 到 0，然后是 LED 的逐行逐列“行军”，最后开始持续滚动显示“Hello Zephyr！”的字样。

该实例展示了较为复杂的单组件运作，从主函数(samples/boards/bbc/microbit/display/src/main.c)可以看到样例通过一个针对 micro:bit 板专用的中间层(drivers/display/mb_display.c)对泛用的显示驱动(头文件 zephyr/drivers/display.h)进行扩充，实现了大多数的功能，例如初始化、打印数字或字母，以及按照 0/1 矩阵点亮 LED 等。

3.2. 闪灯样例和设备树问题

上一节提到，Zephyr 的闪灯样例在 micro:bit V2 上不能运行，本节让我们了解其背后的理由和如何修复与设备树相关的问题。

运行图 14 所示的命令尝试编译闪灯样例：

```
cd ~/zephyrproject/zephyr
west build -b bbc_microbit_v2 samples/basic/blinky -p
```

Figure 14. Commands to compile the blinky sample
图 14. 编译闪灯样例的命令

运行的结果是图 15 所示的编译错误：

Figure 15. Compile error of the blinky sample
图 15. 闪灯样例的编译错误

第 2.2.4 节中提到，Zephyr 提供一整套设备树宏，本例中 GPIO 代码使用的 DT_ALIAS 宏不能完全展开。Zephyr 中设备树宏错误的原因一般都与编译错误中提到的头文件无关，而是设备树有格式/内容的错误，或者访问设备树的方式有误。几种常见的错误如下：

- 混淆了选择、别名、标签名和节点名，或者输入了错误的字符串(例如没有将非字母数字的字符转换为下划线)。
- 在硬件特定的宏中(例如图 15 的 GPIO_DT_SPEC_GET 需要指向一个 GPIO phandle 节点)使用了不同硬件的节点。
- 设备树的节点和绑定的格式要求不一致，导致节点未能生成正确的头文件，因此应用或者驱动中的宏无法展开。注意：这和简单的设备树语法错误不同，语法问题在编译设备树时就会导致编译失败，内容的问题则可能导致在应用代码中无法使用特定属性或宏。
- 使用了错误的宏组合或者宏的参数错误，特别是 For-Each 循环宏和硬件特定的宏。

打开 micro:bit V2 的设备树文件(boards/bbc/microbit_v2/bbc_microbit_v2.dts)，可以看到 aliases 节点下没有 led0，缺少 led0 别名导致了编译的失败[15]。

micro:bit V2 的 LED 矩阵由十个 GPIO 输出控制，个别改变一个控制引脚(pin)并不能点亮 LED。红色的电源指示灯和黄色的 USB 指示灯也并没有连接到 GPIO 上，因此只是依靠开发板本身，我们并不能通过扩展设备树简单地修改好闪灯样例。不过，micro:bit V2 可以外接 LED，将外接 LED 的 GPIO 添加到设备树中就可以修复闪灯样例。

添加设备树覆盖文件 samples/basic/blinky/boards/bbc_microbit_v2.overlay 修复编译错误[16]，见图 16：

```
/ {
    leds {
        compatible = "gpio-leds";
        led0_label: led_0 {
            gpios = <&gpio0 4 GPIO_ACTIVE_HIGH>;
        };
    };

    aliases {
        led0 = &led0_label;
    };
};
```

Figure 16. The device tree overlay file to fix the compilation error
图 16. 修复编译错误的设备树覆盖文件

可以看到文件增加了一个兼容名为 gpio-leds 的节点 leds，然后为含有 GPIO 信息的 led_0 子节点增加别名 led0。gpio-leds 的驱动(drivers/led/led_gpio.c)提供了开关和设定亮度的接口，不过在闪灯样例中，代码(samples/basic/blinky/src/main.c)只是通过 GPIO_DT_SPEC_GET 宏从设备树取得了 GPIO 引脚的信息，然后直接使用 gpio_pin_toggle_dt 切换 GPIO 输出状态。

对比主设备树文件的 edge_connector (边缘连接器)节点和开发板的引脚图[17]可以看到，图 16 中 gpio0 接入点引脚 4 对应 P2 引脚(开发板下侧标记 2 的金手指)，运行时如果有连接外接 LED，闪灯样例就能够运行。

类似的设备树覆盖文件方法，只要正确地修改 GPIO 接入点和引脚号，也可以让没有 led0 别名的开发板支持闪灯样例。

3.3. 样例应用和详解

本节将使用基于官方样例[18]改编的样例应用(https://github.com/lingyuan-he/zephyr-example)。除了主

程序代码还包括：

- 一个简单的自定义代码库(accel)：从 3-轴加速度传感器取得加速度数值，该库可以通过 Kconfig 启用或禁用。
- 一个简单的自定义 LED 矩阵驱动层(ledmatrix)：不使用 Zephyr 的显示驱动，手动通过 GPIO 点亮单个行或列的 LED，该驱动层可以通过 Kconfig 启用/禁用和配置。
- 设备树覆盖文件：用于辅助自定义代码库和 LED 矩阵驱动层，并展示简单的设备树功能。

驱动、代码库和主函数各自配置了日志模块，可以通过 Kconfig 配置日志级别。

3.3.1. 3-轴加速度传感器的代码调用

从主设备树文件上可以看到，micro:bit V2 上内建了 ST 的 lsm303agr 3-轴加速度传感器(见图 17)。在样例应用中，custom-module/lib/accel/accel.c 源代码和 custom-module/include/app/lib/accel.h 头文件将寻找传感器设备和从传感器设备取得 3-轴加速度值的功能包装到了一个简单的自定义库 accel 中。

```
lsm303agr_accel: lsm303agr-accel@19 {
        compatible = "st,lis2dh", "st,lsm303agr-accel";
        status = "okay";
        reg = <0x19>;
        irq-gpios = <&gpio0 25 GPIO_ACTIVE_HIGH>;
};
```

Figure 17. micro:bit V2 device tree file snippet (source: Zephyr on GitHub)
图 17. micro:bit V2 设备树文件片段(来源：Zephyr GitHub 代码库)

accel 库代码中，寻找传感器设备的 get_accel_device 函数通过别名 accel 寻找设备树中的加速度传感器设备，这一别名在 micro:bit V2 的主设备树文件中并不存在(其中只有 accel0)，而是由样例应用设备树覆盖文件(app/boards/bbc_microbit_v2.overlay)提供的。其中增加了 accel 别名，指向标签为 lsm303agr_accel 的节点。

设备树覆盖文件能在开发板的主设备树文件上进行增添和修改，它的几项用途如下[19]：

- 增加别名(本例的 accel)或者选择。
- 覆写已有节点的属性值，例如更改串口的数据速率。
- 删除节点的一个属性。
- 增加子节点，例如总线上新的子设备。

回到 accel.c 代码中，get_accel_values 函数用于获取 3-轴加速度值，其中 sensor_sample_fetch 和 sensor_channel_get 函数调用完成了样本刷新和取样本值的功能。了解它们是如何针对特定的传感器完成代码调用的，能够帮助我们更加深入地理解 Zephyr 的设备驱动模型(第 2.1 节)。

sensor_sample_fetch 和 sensor_channel_get 函数均为泛用传感器驱动 API，从 Zephyr 代码库头文件 include/zephyr/drivers/sensor.h 可以看到两个函数会分别调用设备驱动 API sample_fetch 和 channel_get 函数。设备树中设备的兼容名决定了适配的驱动程序。在设备树文件中，传感器的兼容名有两个：“st,lis2dh”和“st,lsm303agr-accel”。驱动的适配顺序是先查找第一个兼容名，在 Zephyr 代码中搜索 st_lis2dh (非字母数字的字符替代为下划线)，可以找到 drivers/sensor/st/lis2dh/lis2dh.c 文件包含定义驱动的语句“#define DT_DRV_COMPAT st_lis2dh”。图 18 所示的是该驱动的驱动 API 结构定义：

```
static DEVICE_API(sensor, lis2dh_driver_api) = {
        .attr_set = lis2dh_attr_set,
#if CONFIG_LIS2DH_TRIGGER
        .trigger_set = lis2dh_trigger_set,
#endif
        .sample_fetch = lis2dh_sample_fetch,
        .channel_get = lis2dh_channel_get,
};
```

Figure 18. lis2dh device driver API definition (source: Zephyr on GitHub)
图 18. lis2dh 设备驱动的 API 定义(来源：Zephyr GitHub 代码库)

可以看到该驱动将 lis2dh_sample_fetch 和 list2dh_channel_get 函数的指针指定为设备 sample_fetch 和 channel_get API 的实现。lis2dh 驱动支持 I2C 和 SPI 总线，在主设备树文件中可以看到，micro:bit V2 中的传感器是在 i2c 总线上的。图 19 所示的是 lis2dh 驱动的部分初始化代码：

```
#define LIS2DH_CONFIG_I2C(inst)                                         \
        {                                                               \
                .bus_init = lis2dh_i2c_init,                            \
                .bus_cfg = { .i2c = I2C_DT_SPEC_INST_GET(inst), },      \
                .hw = { .is_lsm303agr_dev = IS_LSM303AGR_DEV(inst),     \
                        .disc_pull_up = DISC_PULL_UP(inst),             \
                        .anym_on_int1 = ANYM_ON_INT1(inst),             \
                        .anym_latch = ANYM_LATCH(inst),                 \
                        .anym_mode = ANYM_MODE(inst), },                \
                LIS2DH_CFG_TEMPERATURE(inst)                            \
                LIS2DH_CFG_INT(inst)                                    \
        }

#define LIS2DH_DEFINE_I2C(inst)                                         \
        static struct lis2dh_data lis2dh_data_##inst;                   \
        static const struct lis2dh_config lis2dh_config_##inst =        \
                LIS2DH_CONFIG_I2C(inst);                                \
        LIS2DH_DEVICE_INIT(inst)
/*
 * Main instantiation macro. Use of COND_CODE_1() selects the right
 * bus-specific macro at preprocessor time.
 */

#define LIS2DH_DEFINE(inst)                                             \
        COND_CODE_1(DT_INST_ON_BUS(inst, spi),                          \
                    (LIS2DH_DEFINE_SPI(inst)),                          \
                    (LIS2DH_DEFINE_I2C(inst)))

DT_INST_FOREACH_STATUS_OKAY(LIS2DH_DEFINE)
```

Figure 19. lis2dh device driver initialization code (source: Zephyr on GitHub)
图 19. lis2dh 设备驱动初始化代码(来源：Zephyr GitHub 代码库)

代码通过 DT_INST_FOREACH_STATUS_OKAY 宏，对每一个状态为 okay 的兼容设备扩展

LIS2DH_DEFINE 宏，后者会通过 DT_INST_ON_BUS 判断设备是否在 spi 总线上，如果是，就进一步扩展 LIS2DH_DEFINE_SPI 初始化驱动，否则会扩展 LIS2DH_DEFINE_I2C 宏(micro:bit V2 的情况)。那么，设备树是如何让 DT_INST_ON_BUS 能够进行判定的呢？

micro:bit V2 设备树中传感器所在的 i2c 节点兼容名为“nordic,nrf-twim”，从其绑定文件 dts/bindings/i2c/nordic,nrf-twim.yaml 中可以看到，文件包含(include)了 nordic,nrf-twi-common.yaml (同文件夹下)，然后该文件又进一步包含了 i2c-controller.yaml，在这一文件中终于看到了“bus: i2c”的信息。也就是说，从设备树绑定可以得知传感器从属于使用 i2c 总线的控制器。

由于 lis2dh 驱动能够被正确地配置，系统不会查找兼容“st,lsm303agr-accel”的驱动。在运行时，accel 代码库中的 sensor_sample_fetch 和 sensor_channel_get 函数会调用 st_lis2dh 驱动的函数。

在 Zephyr 的在线文档中，通过兼容名可以找到设备树绑定的参考页面，例如本例中的驱动文档标题为“st,lis2dh (on i2c bus)”。

3.3.2. 设备树绑定和自定义驱动

在样例应用中，自定义的 ledmatrix 驱动(custom-module/drivers/ledmatrix/ledmatrix.c)使用 GPIO 在 LED 矩阵上实现了简单点亮矩阵边缘一排或一行 5 枚 LED 的功能。在前一节中提到，驱动需要匹配到设备树的设备节点上。本例中我们创建了自定义的“custom-ledmatrix”兼容名和其绑定，以及 ledmatrix 驱动实现。

图 20 和图 21 所示的分别是 custom-ledmatrix 设备树绑定文件(custom-module/dts/bindings/custom-ledmatrix.yaml)和 micro:bit V2 设备树覆盖文件中的对应节点：

```
description: |
  An LED matrix that consists of row and column GPIO pins.

# Device tree node will have the same "compatible" attribute to be picked up
# by a driver that intends to operate on the device.
compatible: "custom-ledmatrix"

include: base.yaml

properties:
  led-row-gpios:
    type: phandle-array
    required: true
    description: |
      Array of row GPIOs pins that are part of the matrix.

  led-col-gpios:
    type: phandle-array
    required: true
    description: |
      Array of column GPIOs pins that are part of the matrix.
```

Figure 20. The device tree binding file for custom-ledmatrix
图 20. custom-ledmatrix 设备树绑定文件

```
/*
 * Define a node of the custom "custom-ledmatrix" binding that contains GPIO
 * information from the 5x5 LED matrix. Provide the same GPIO pins as the
 * "led_matrix" node in the bbc_microbit_v2 board devicetree source file.
 */
ledmatrix_label: custom_ledmatrix {
        compatible = "custom-ledmatrix";
        status = "okay";
        led-row-gpios = <&gpio0 21 GPIO_ACTIVE_HIGH>,
                        <&gpio0 22 GPIO_ACTIVE_HIGH>,
                        <&gpio0 15 GPIO_ACTIVE_HIGH>,
                        <&gpio0 24 GPIO_ACTIVE_HIGH>,
                        <&gpio0 19 GPIO_ACTIVE_HIGH>;
        led-col-gpios = <&gpio0 28 GPIO_ACTIVE_LOW>,
                        <&gpio0 11 GPIO_ACTIVE_LOW>,
                        <&gpio0 31 GPIO_ACTIVE_LOW>,
                        <&gpio1 5 GPIO_ACTIVE_LOW>,
                        <&gpio0 30 GPIO_ACTIVE_LOW>;
};
```

Figure 21. The custom-ledmatrix device tree node
图 21. cutstom-ledmatrix 设备树节点

从图 20 中可以看到，custom-ledmatrix 绑定中有两个 GPIO 引脚 phandle 数组，分别代表 LED 矩阵的行 GPIO 引脚(推挽)和列 GPIO 引脚(开漏) [20]。在图 21 中，注意到 GPIO 接入点、引脚号和逻辑电平模式与开发板主设备树文件中“led_matrix”节点(兼容名“nordic,nrf-led-matrix”)是一致的[21]。样例中我们使用 GPIO 在不使用动态刷新的情况下进行亮、灭灯，所以不需要其他的属性。

需要特别注意的是，为了表示 phandle 每个说明符(specifier)成员的长度(例如 GPIO 除了接入点之外需要提供两个数据成员)，在绑定中一般应提供名称为“#*-cells”的属性。不过由于 GPIO 类 phandle 十分常见，只要属性的命名以“-gpios”结尾，如本例中的 led-row-gpios 和 led-col-gpios，就不需要提供这一属性。关于“#*-cells”属性的细节详见官方文档。

从图 21 中还可以看到设定设备状态就绪的语句(status 为“okay”)，节点能够使用该属性是因为绑定文件包含了 base.yaml。上一节中提到，标记设备就绪对于驱动的初始化是必须的，例如图 19 中用到的 DT_INST_FOREACH_STATUS_OKAY 宏。

自定义驱动的头文件定义见 custom-module/include/app/drivers/ledmatrix.h，可以看到驱动 API 由 5 个函数组成(见 ledmatrix_driver_api 结构定义)，分别负责点亮 LED 矩阵最边缘的行或是列(共 4 个 API)和关闭 LED 显示(第 5 个 API)。在驱动的实现(custom-module/drivers/ledmatrix/ledmatrix.c)中，这 5 个函数会被实现(见 driver_api 结构) [22]。现在读者应该能够理解 ledmatrix 驱动的基本结构。最后，图 22 所示的是驱动的初始化宏：

```
#define LEDMATRIX_DEFINE(inst)                                          \
                                                                        \
    static const struct ledmatrix_config config##inst = {              \
        .rows = { DT_INST_FOREACH_PROP_ELEM_SEP(inst, led_row_gpios,    \
                                        GPIO_DT_SPEC_GET_BY_IDX, (, )) },   \
        .cols = { DT_INST_FOREACH_PROP_ELEM_SEP(inst, led_col_gpios,    \
                                        GPIO_DT_SPEC_GET_BY_IDX, (, )) },   \
    };                                                                  \
                                                                        \
    DEVICE_DT_INST_DEFINE(inst, instance_init, NULL, NULL, &config##inst,   \
                          POST_KERNEL, CONFIG_LEDMATRIX_INIT_PRIORITY,  \
                          &driver_api);

DT_INST_FOREACH_STATUS_OKAY(LEDMATRIX_DEFINE)
```

Figure 22. The initialization of the ledmatrix driver
图 22. ledmatrix 驱动的初始化

LEDMATRIX_DEFINE 中使用 GPIO_DT_SPEC_GET_BY_IDX 配合 DT_INST_FOREACH_PROP_ELEM_SEP，从设备树循环提取 GPIO 引脚 phandle 数组中的成员，从而静态组成 gpio_dt_spec 数组[23]，用于在设备驱动配置结构(见图 23)中存储行和列 GPIO 引脚属性[24]。

```
struct ledmatrix_config {
    struct gpio_dt_spec rows[NUM_ROW];
    struct gpio_dt_spec cols[NUM_COL];
};
```

Figure 23. The configuration structure of the ledmatrix driver
图 23. ledmatrix 驱动的配置结构

和上一节提到的传感器驱动类似，DT_INST_FOREACH_STATUS_OKAY 针对每个状态为就绪的、兼容名为“custom-ledmatrix”的设备进行驱动初始化。

通过 ledmatrix 驱动层，样例应用的主函数就可以很容易地直接进行 LED 行或是列的点亮操作。配合加速度传感器的数据，样例应用实现了根据重力方向点亮 LED 矩阵对应边缘行或列的效果。

3.3.3. 日志系统

Zephyr 提供了日志系统的支持，应用代码、驱动、代码库可以注册各自的日志模块，并通过 Kconfig 配置模块的日志级别。日志的可能级别从低到高分别为：DBG (调试)、INF (信息)、WRN (警告)和 ERR (错误)。代码中通过调用 LOG_X (X 为级别)宏就可以使用与 printk 类似的语法写日志。以样例应用中的 accel 代码库为例，custom-module/lib/accel/accel.c 中包含了 zephyr/logging/log.h 头文件，然后使用 LOG_MODULE_REGISTER 宏定义了日志模块 accel，其日志级别为 CONFIG_ACCELLIB_LOG_LEVEL。

在 Kconfig 中，CONFIG_LOG 配置用于在全局启用日志，然后通过添加 CONFIG_<模块>_LOG_LEVEL_X (X 为级别)配置设定个别模块的级别。本例中应用的配置文件 app/prj.conf 通过 CONFIG_LOG=y 在全局开启了日志功能，然后通过 CONFIG_ACCELLIB_LOG_LEVEL_INF=y 选项将 accel 模块的日志级别定义为 INF(信息)级别。ledmatrix 驱动和应用主代码各自也有日志模块的配置。

使用日志系统相比使用 printk 更加可配置，例如只有调试时才需要的日志可以通过默认日志级别进行过滤，发布应用时也可以很容易地禁用日志输出。

4. 运行和调试 Zephyr 应用

4.1. 运行样例应用

编译和部署样例应用的命令如图 24 所示：

```
cd ~/zephyrproject
mkdir applications && cd applications
git clone https://github.com/lingyuan-he/zephyr-example.git
cd zephyr-example
west build -b bbc_microbit_v2 app -p
west flash
```

Figure 24. Commands to download and deploy the example application
图 24. 下载和部署样例应用的命令

样例应用开始运行时，将开发板平放于台面上，此时 LED 矩阵不会点亮，如果将开发板拿起，一侧垂直朝向地面时，检测到重力一侧的一排或一列 5 个 LED 会点亮。例如，当开发板垂直于台面正面并面

向读者时，LED 矩阵最下一行会点亮(见图 25)。

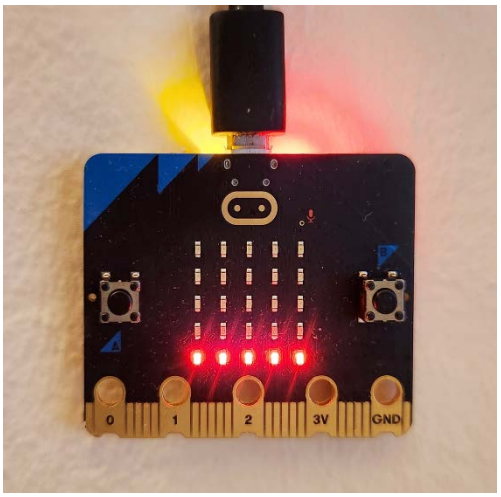

Figure 25. Illumination of the bottom row LEDs when the board is upright
图 25. 开发板垂直摆放时，最下一排的 LED 点亮

4.2. Zephyr 应用的调试

在 Zephyr 应用开发中，最简单的调试方法就是输出日志。micro:bit V2 运行样例应用时会将日志输出到串口，可以通过任何串口工具连接串口，例如使用 minicom 的命令：“minicom -D /dev/ttyACM0 -b 115200”。

样例应用的默认日志级别为 INF，编译时可以通过包括 debug.conf 的选项将日志级别降低为 DBG，程序就会输出传感器数据和 GPIO 操作细节，命令为：“west build -b bbc_microbit_v2 app -p --extra-conf debug.conf”。

Zephyr 支持在 micro:bit V2 上使用 GDB 进行远程调试，应用编译和烧录(“west build”和“west flash”)后，运行“west debug”就会启动 GDB。GDB 简单的用法例如：设置断点(“b main.c:<行数>”或“b <函数名>”)、逐行执行(n)、继续执行(c)和打印变量(“p <变量名>”)。“west debug”命令还可以指定 GDB 以外的调试接口[25]，例如 jlink 和 openocd。

5. 结语

Zephyr 在系统设计上借鉴了 Linux 等大型开源软件的设计理念，引入了 Linux 和桌面系统开发者熟悉的概念和开发过程，但相对常见的 RTOS，复杂度增加了数个级别。通过将硬件进行抽象化，以及提供帮助简化开发过程的工具和框架(例如 west 工具和 twister 测试框架)，Zephyr 希望能吸引不同领域的开发者和企业用户。但是，开发和调试难度的上升也让不少开发者望而却步，特别是熟悉面向硬件直接编程或是使用小型 RTOS 的嵌入式开发者。

希望在阅读本文后，读者对在 Zephyr 上进行嵌入式软件开发有了初步的了解。本文中的实例并不涉及过于具体的硬件细节或是复杂的应用需求，Zephyr 的官方文档、实例，以及北欧半导体等硬件厂商的样例项目都十分有参考价值。虽然官方文档的中文化有所欠缺，但国内开发者在各类平台上发布的学习笔记一直在增加，线上讨论也十分热烈。

Zephyr 近年来劲头强势，硬件厂商、开发者和开源社区的热情正盛，项目的开发活跃程度远超其他 RTOS。期待 Zephyr 项目在未来能够简化复杂的系统架构，改善学习难度高和代码调试困难等问题，并

覆盖更多的硬件和应用，成为一个全方位的主流物联网操作系统。

参考文献

[1] Zephyr Project (2025) About the Zephyr Project. https://zephyrproject.org/learn-about/

[2] Eclipse Foundation (2024) 2024 IoT & Embedded Developer Survey Report. https://outreach.eclipse.foundation/iot-embedded-developer-survey-2024

[3] 王洪波. 嵌入式虚拟化技术与应用: ACRN 开源项目实践[M]. 北京: 机械工业出版社, 2023.

[4] Zephyr Project (2024) Zephyr Project Overview. https://www.zephyrproject.org/wp-content/uploads/2024/10/Zephyr-Overview-20241017.pdf

[5] Zephyr Project (2025) Device Driver Model. https://docs.zephyrproject.org/latest/kernel/drivers/index.html

[6] The Devicetree Organization (2023) Devicetree Specification, Release v0.4. https://github.com/devicetree-org/devicetree-specification/releases/download/v0.4/devicetree-specification-v0.4.pdf

[7] Zephyr Project (2025) Devicetree. https://docs.zephyrproject.org/latest/build/dts/index.html

[8] Zephyr Project (2025) Scope and Purpose. https://docs.zephyrproject.org/latest/build/dts/intro-scope-purpose.html

[9] Zephyr Project (2025) Syntax and Structure. https://docs.zephyrproject.org/latest/build/dts/intro-syntax-structure.html

[10] Zephyr Project (2025) Devicetree Bindings. https://docs.zephyrproject.org/latest/build/dts/bindings.html

[11] Lampacher, M. (2025) Practical Zephyr Git Repository. https://github.com/lmapii/practical-zephyr/tree/main/03_devicetree_semantics

[12] Eliasz, A. (2024) Zephyr RTOS Embedded C Programming. Apress Berkeley.

[13] Lampacher, M. (2024) Practical Zephyr-Kconfig (Part 2). https://interrupt.memfault.com/blog/practical_zephyr_kconfig

[14] The Micro:Bit Organization (2025) Meet the New BBC Micro:Bit. https://microbit.org/new-microbit/

[15] Gammell, C. (2024) Zephyr for Hardware Engineers: GPIO. https://blog.golioth.io/zephyr-for-hardware-engineers-gpio/

[16] Valens, C. (2024) Getting Started with the Zephyr RTOS. *Elektor*, **2**, 98-105.

[17] The Micro:Bit Organization (2025) Micro:Bit Pins. https://makecode.microbit.org/device/pins

[18] Zephyr Project (2025) Example Application Git Repository. https://github.com/zephyrproject-rtos/example-application/tree/main

[19] Zephyr Project (2025) Devicetree HOWTOs. https://docs.zephyrproject.org/latest/build/dts/howtos.html

[20] Zephyr Project (2021) Zephyr and the BBC Microbit V2 Tutorial Part 1: GPIO. https://www.zephyrproject.org/zephyr-and-the-bbc-microbit-v2-tutorial-part-1-gpio/

[21] Lampacher, M. (2024) Practical Zephyr-Devicetree Semantics (Part 4). https://interrupt.memfault.com/blog/practical_zephyr_dt_semantics

[22] Szczys, M. (2024) How to Write a Zephyr Device Driver with a Custom API. https://blog.golioth.io/how-to-write-a-zephyr-device-driver-with-a-custom-api/

[23] Nordic Semiconductor (2025) GPIO Generic API. https://academy.nordicsemi.com/courses/nrf-connect-sdk-fundamentals/lessons/lesson-2-reading-buttons-and-controlling-leds/topic/gpio-generic-api/

[24] Lampacher, M. (2024) Practical Zephyr-Devicetree Practice (Part 5). https://interrupt.memfault.com/blog/practical_zephyr_05_dt_practice

[25] Zephyr Project (2025) Building, Flashing and Debugging. https://docs.zephyrproject.org/latest/develop/west/build-flash-debug.html