第 3 章

移植 MicroPython 最小工程

MicroPython 提供了 minimal 工程，它是基于微控制器平台开发的起点。本章将在现有的 MM32F3 微控制器平台上移植 minimal 工程并将之下载到 MM32F3 的开发板上，利用 UART 串口作为 REPL，在 MM32F3 微控制器上运行 MicroPython 内核来执行 Python 脚本。在移植 minimal 工程的过程中，也将为后续增加硬件外设相关的类模块做好规划，约定好后续添加源代码的规则。

3.1 MicroPython 的最小工程

原始的 MicroPython 提供了两个最小工程：bare-arm 和 minimal。bare-arm 相对于 minimal 更小，它以最简单直接的方式实现了一个完整的 Arm 内核微控制器平台上的 MicroPython 移植。bare-arm 工程的移植代码及其 Makefile 都是最简单的，对于想深入研究 MicroPython 内核的开发者有比较高的价值，这个工程以最小的代码量同一个可运行的微控制器平台建立关联，可以让开发者集中精力调试 MicroPython 内核，尽量减少在编译具体微控制器平台相关源码上所花费的时间。

对于关注应用的开发者，通常会将开发和研究的重心放在具体的微控制器平台上，而非实现 Python 功能的内核。作者参考 MicroPython 社区开发者的建议和自己的开发经验，选择 minimal 作为基于平台进行开发的起点。相对于 bare-arm，minimal 工程的组织结构对在微控制器平台上实现各部分功能的源代码进行了明确的划分，并且引入了 MicroPython 的存储管理组件 GC，它更接近一个成熟的移植项目。基于一个具体的微控制器平台，以 minimal 工程为起点，只要继续新增与微控制器外设和电路板相关的类模块即可。

3.1.1 minimal 项目目录下的文件

minimal 项目目录下包含基于微控制器 STM32F405 且支持 MicroPython 的基本移植源代码和项目组织文件，见表 3-1。

表 3-1 minimal 项目目录下的文件

文 件 名	功 能 简 要 说 明
frozentest.mpy/ frozentest.py	Python 预编译功能的测试代码
main.c	minimal 项目的 main()函数所在文件

续表

文 件 名	功 能 简 要 说 明
Makefile	minimal 项目组织文件，通过 make 启动编译过程
mpconfigport. h	移植过程中对 MicroPython 的配置文件，主要配置 MicroPython 内核
mphalport. h	移植过程中对底层硬件的配置文件，目前仅适配了一个空的时钟服务的底层实现，后续会适配引脚
qstrdefsport. h	人工定义 MicroPython 中的 QSTR 字符关键字清单，同 MicroPython 编译过程中从源代码中扫描提取的 QSTR 整合，生成最终的 QSTR 源文件交给 C 编译器
stm32f405. ld	基于具体微控制器平台 STM32F405 的链接命令文件，用于管理可执行程序中的内存分布。按照 gcc 编译工具链需要的格式编写
uart_core. c	实现通过具体微控制器的 UART 对接 REPL 的源程序

3.1.1.1 微控制器平台最小工程

main. c、uart_core. c 和 stm32f407. ld 文件与微控制器平台紧密相关，可直接访问微控制器寄存器或者调用固件库。这部分源代码是嵌入式系统软件工程师熟悉的内容，通常也能在微控制器的固件库中找到对应的内容。在 uart_core. c 中，更是可以直接操作微控制器硬件寄存器，并定义了 REPL 使用的串口的相关操作，见代码 3-1。

代码 3-1 uart_core. c 文件中绑定 REPL 的串口

```
/*
 * Core UART functions to implement for a port
 */

#if MICROPY_MIN_USE_STM32_MCU
typedef struct {
    volatile uint32_t SR;
    volatile uint32_t DR;
} periph_uart_t;
#define USART1 ((periph_uart_t *)0x40011000)
#endif

// 接收单个字符
int mp_hal_stdin_rx_chr(void) {
    unsigned char c = 0;
    #if MICROPY_MIN_USE_STDOUT
    int r = read(STDIN_FILENO, &c, 1);
    (void)r;
    #elif MICROPY_MIN_USE_STM32_MCU
    // wait for RXNE
    while ((USART1->SR & (1 << 5)) == 0) {
    }
    c = USART1->DR;
    #endif
    return c;
}
```

```
// 发送指定长度的字符串
void mp_hal_stdout_tx_strn(const char * str, mp_uint_t len) {
    #if MICROPY_MIN_USE_STDOUT
    int r = write(STDOUT_FILENO, str, len);
    (void)r;
    #elif MICROPY_MIN_USE_STM32_MCU
    while (len--) {
        // wait for TXE
        while ((USART1->SR & (1 << 7)) == 0) {
        }
        USART1->DR = * str++;
    }
    #endif
}
```

main.c 文件也包含了 STM32F405 的中断向量表和复位向量函数的定义，与 stm32f407.ld 文件一起构建了 STM32F405 微控制器的最小工程。其中，定义微控制器复位上电的程序执行流程的代码，见代码 3-2。

代码 3-2　minimal 中 main.c 定义的微控制器复位程序执行流程

```
#if MICROPY_MIN_USE_CORTEX_CPU

// 这里使用了 Cortex-M 内核处理器最简单的上电复位操作流程
extern uint32_t _estack, _sidata, _sdata, _edata, _sbss, _ebss;

void Reset_Handler(void) __attribute__((naked));
void Reset_Handler(void) {
    // 设定栈顶指针
    __asm volatile ("ldr sp, =_estack");
    // 将.data 段内容从 Flash 搬运到 RAM
    for (uint32_t * src = &_sidata, * dest = &_sdata; dest < &_edata;) {
        * dest++ = * src++;
    }
    // 将.bss 段的 RAM 空间清零
    for (uint32_t * dest = &_sbss; dest < &_ebss;) {
        * dest++ = 0;
    }
    // 跳转到用户定义的启动函数
    void _start(void);
    _start();
}

void Default_Handler(void) {
    for (;;) {
    }
}

// Cortex-M 处理器内核的中断向量表
```

```
const uint32_t isr_vector[] __attribute__((section(".isr_vector"))) = {
    (uint32_t)&_estack,
    (uint32_t)&Reset_Handler,
    (uint32_t)&Default_Handler, // NMI_Handler
    (uint32_t)&Default_Handler, // HardFault_Handler
    (uint32_t)&Default_Handler, // MemManage_Handler
    (uint32_t)&Default_Handler, // BusFault_Handler
    (uint32_t)&Default_Handler, // UsageFault_Handler
    0,
    0,
    0,
    0,
    (uint32_t)&Default_Handler, // SVC_Handler
    (uint32_t)&Default_Handler, // DebugMon_Handler
    0,
    (uint32_t)&Default_Handler, // PendSV_Handler
    (uint32_t)&Default_Handler, // SysTick_Handler
};

void _start(void) {
// 当执行至此,CPU 系统的堆栈都已经初始化完毕,bss 段已清零,data 段已赋初值
    // SCB->CCR: enable 8-byte stack alignment for IRQ handlers, in accord with EABI
    *((volatile uint32_t *)0xe000ed14) |= 1 << 9;

    // 初始化硬件外设
    #if MICROPY_MIN_USE_STM32_MCU
    void stm32_init(void);
    stm32_init();
    #endif

    // 至此,已经准备好运行环境,并可进入用户定义的 main()函数
    main(0, NULL);

    // 不应该执行到这里
    for (;;) {
    }
}

#endif
```

3.1.1.2 在 main()中启动 MicroPython

main.c 中的 main()函数实现了启动 MicroPython 作为一个大应用程序的初始化和执行过程,见代码 3-3。

代码 3-3 minimal 项目中 main()函数启动 MicroPython

```
static char *stack_top;
#if MICROPY_ENABLE_GC
```

```
static char heap[2048];
#endif

/* main()函数,应用程序入口 */
int main(int argc, char * * argv)
{
    int stack_dummy;
    stack_top = (char * )&stack_dummy;

    #if MICROPY_ENABLE_GC
    gc_init(heap, heap + sizeof(heap));
    #endif
    mp_init();
    #if MICROPY_ENABLE_COMPILER
    #if MICROPY_REPL_EVENT_DRIVEN
    pyexec_event_repl_init();
    for (;;) {
        int c = mp_hal_stdin_rx_chr();
        if (pyexec_event_repl_process_char(c)) {
            break;
        }
    }
    #else
    pyexec_friendly_repl();
    #endif
    // do_str("print('hello world!', list(x+1 for x in range(10)), end='eol\\n')",
MP_PARSE_SINGLE_INPUT);
    // do_str("for i in range(10):\r\n  print(i)", MP_PARSE_FILE_INPUT);
    #else
    pyexec_frozen_module("frozentest.py");
    #endif
    mp_deinit();
    return 0;
}
```

从源代码中可以看到,minimal项目的执行流程如下:

(1) 初始化内存管理器,调用gc_init()指定堆空间。GC是MicroPython的内存管理组件,实例化对象创建的内存,都是存放在GC管辖的内存中。顾名思义,在MicroPython认为的特定时机,也会回收不再使用的变量所占用的内存。

(2) 初始化MicroPython内核,调用mp_init()。

(3) 初始化REPL,调用pyexec_event_repl_init()。

(4) 之后就是在一个无限循环中从UART串口接收数据,解析,然后执行。或者在pyexec_friendly_repl()函数中做类似的人机交互。再或者,直接执行frozentest.py脚本。这里的MICROPY_ENABLE_GC、MICROPY_ENABLE_COMPILER是在mpconfigport.h中指定的,见代码3-4。

代码 3-4　mpconfigport.h 中关于解释器和 GC 的配置

```
#define MICROPY_ENABLE_COMPILER          (1)
#define MICROPY_ENABLE_GC                (1)
```

MICROPY_REPL_EVENT_DRIVEN 没有出现在 mpconfigport.h 中，这里使用的是默认值，在源文件 py/mpconfig.h 中指定，见代码 3-5。

代码 3-5　mpconfig.h 中关于 REPL 的配置

```
// Whether port requires event-driven REPL functions
#ifndef MICROPY_REPL_EVENT_DRIVEN
#define MICROPY_REPL_EVENT_DRIVEN (0)
#endif
```

所以，这里的 main.c 实际执行的是 pyexec_friendly_repl()，这个函数位于 py/pyexec.c 文件中，见代码 3-6。

代码 3-6　pyexec.c 中的 pyexec_friendly_repl()

```
int pyexec_friendly_repl(void) {
    vstr_t line;
    vstr_init(&line, 32);

friendly_repl_reset:
    mp_hal_stdout_tx_str("MicroPython " MICROPY_GIT_TAG " on " MICROPY_BUILD_DATE ";
" MICROPY_HW_BOARD_NAME " with " MICROPY_HW_MCU_NAME "\r\n");
    #if MICROPY_PY_BUILTINS_HELP
    mp_hal_stdout_tx_str("Type \"help()\" for more information.\r\n");
    #endif
    ...
    vstr_reset(&line);
    int ret = readline(&line, ">>> ");
    ...
            // got a line with non-zero length, see if it needs continuing
            while (mp_repl_continue_with_input(vstr_null_terminated_str(&line))) {
                vstr_add_byte(&line, '\n');
                ret = readline(&line, "... ");
                if (ret == CHAR_CTRL_C) {
                    // cancel everything
                    mp_hal_stdout_tx_str("\r\n");
                    goto input_restart;
                } else if (ret == CHAR_CTRL_D) {
                    // stop entering compound statement
                    break;
                }
            }
    ...
        ret = parse_compile_execute(&line, parse_input_kind, EXEC_FLAG_ALLOW_DEBUGGING |
EXEC_FLAG_IS_REPL | EXEC_FLAG_SOURCE_IS_VSTR);
```

```
            if (ret & PYEXEC_FORCED_EXIT) {
                return ret;
            }
    }
```

从代码 3-6 中可以看到，每次复位开发板后，在 REPL 中显示的一串包含版本号和开发板信息的字符串，就是在这里输出的。之后，通过 vstr 相关函数逐行获取 REPL 的输入字符串，最后放到 parse_compile_execute()函数中，执行 MicroPython 的解释执行过程。parse_compile_execute()位于 lib/utils/pyexec.c 文件中。从这个函数深入研究下去，就可以研究 MicroPython 逐步解析 Python 语句并执行的细节了。这将会走进 Python 内核，但同微控制器平台移植关联不大，因此这里暂不深究。

代码 3-6 还显现出对 help()函数的调用，如果启用 MICROPY_PY_BUILTINS_HELP（在 py/mpconfig.h 中默认禁用，但在 minimal 项目目录下的 mpconfigport.h 文件中打开），则还会继续输出关于 help()函数的提示信息。追溯宏选项“MICROPY_PY_BUILTINS_HELP”的定义，在 mpconfig.h 文件中，还可以看到更多关于 help()函数相关的功能，见代码 3-7。

代码 3-7　mpconfig.h 文件中关于 help()的宏选项

```
// Whether to provide the help function
#ifndef MICROPY_PY_BUILTINS_HELP
#define MICROPY_PY_BUILTINS_HELP (0)
#endif

// Use this to configure the help text shown for help().  It should be a
// variable with the type "const char * ".  A sensible default is provided.
#ifndef MICROPY_PY_BUILTINS_HELP_TEXT
#define MICROPY_PY_BUILTINS_HELP_TEXT mp_help_default_text
#endif

// Add the ability to list the available modules when executing help('modules')
#ifndef MICROPY_PY_BUILTINS_HELP_MODULES
#define MICROPY_PY_BUILTINS_HELP_MODULES (0)
#endif
```

例如，宏开关 MICROPY_PY_BUILTINS_HELP_MODULES 可以启用支持类模块的 help()函数，而 MICROPY_PY_BUILTINS_HELP_TEXT 指定了单纯输入 help()语句时输出的提示字符串为 py/buildinhelp.c 中定义的 mp_help_default_text，其实现见代码 3-8。关于更多 MicroPython 内建 help()函数的相关实现可见于 py/buildinhelp.c 文件，或者通过这两个宏开关在 MicroPython 的源码目录中追溯，这里暂不深究。

代码 3-8　buildinhelp.c 文件中的 help()提示字符串

```
const char mp_help_default_text[] =
    "Welcome to MicroPython!\n"
```

```
        "\n"
        "For online docs please visit http://docs.micropython.org/\n"
        "\n"
        "Control commands:\n"
        "  CTRL-A        -- on a blank line, enter raw REPL mode\n"
        "  CTRL-B        -- on a blank line, enter normal REPL mode\n"
        "  CTRL-C        -- interrupt a running program\n"
        "  CTRL-D        -- on a blank line, exit or do a soft reset\n"
        "  CTRL-E        -- on a blank line, enter paste mode\n"
        "\n"
        "For further help on a specific object, type help(obj)\n"
    ;
```

3.1.1.3 3个带有port后缀的源文件

除main.c文件之外，还需要具体微控制器芯片和开发板相关的，涉及移植操作的代码，它们主要位于mpconfigport.h、mphalport.h和qstrdefsport.h这3个带有port后缀的源文件中：

- mpconfigport.h用于在移植项目中配置MicroPython的内核功能，其中包含了在编译过程中启动MicroPython子功能的宏开关。与其说是归属于MicroPython的移植，作者更倾向于认为它是对MicroPython内核功能的裁剪，主要作用于MicroPython内核，对硬件依赖性很少。在基于MM32F3微控制器的移植中，将详细介绍实际使用的宏开关对应的功能。
- mphalport.h承担了一部分与硬件相关的移植工作。

目前这个源文件里仅仅实现了两个空函数，是为了保证整个项目能够正常编译而创建的两个“占位函数”，见代码3-9。

代码3-9 mphalport.h中同硬件移植相关的函数

```
static inline mp_uint_t mp_hal_ticks_ms(void) {
    return 0;
}
static inline void mp_hal_set_interrupt_char(char c) {
}
```

在后面的移植过程中还会用到这个文件，这个文件定义的函数将间接调用关于引脚和定时器的硬件驱动函数，为其他多个模块提供关于引脚和定时器的部分服务。

- qstrdefsport.h中将保存人为指定的MicroPython的QSTR字符串。

MicroPython使用QSTR引用对象实体（变量、函数、类模块等都是对象）。如果使用MicroPython自带的Makefile文件执行编译，将新增的包含QSTR关键字定义的源文件加入到变量SRC_QSTR中，则在编译期间会自动调用Python脚本（py/makemoduledefs.py、/py/makeqstrdefs.py）从参与编译的源程序代码中提取QSTR关键字清单；否则，需要开发者自行在本文件中添加QSTR。得益于make工具对编译过程的灵活控制，能够在编译过程中插入自定义的预处理脚本自动处理QSTR的提取工作。对于一些集成开发环境，可

能未向用户开放自定义预处理脚本的编程接口，此时就需要人工收集 QSTR 关键字到这个文件了。

作者在早期的学习过程中，曾尝试在 qstrdefsports.h 文件中人工定义 QSTR 关键字清单。当时，参考 MicroPython 的开发者手册和已有其他平台的项目，编写源代码创建了一个新模块 led 用以点亮小灯，将该模块添加到移植的 MicroPython 工程后，可以编译通过，但是在执行可执行文件时，就是不能在 REPL 中调用预先定义好的类方法 on 和 off(但能引用类名 led)。当时大体定位是 QSTR 未识别的问题，但尚未找到合适的处理流程。后来经过一系列调试，在 qstrdefsports.h 文件中添加了 QSTR 的定义，再编译运行，就可以在 REPL 中识别到 led 模块的 on 和 off 方法了，见代码 3-10。

代码 3-10　在 qstrdefsports.h 文件中人工添加 QSTR 字符串

```
// qstrs specific to this port
// * FORMAT - OFF *

Q(on)
Q(off)
```

当然，正确的做法是，将新增的包含 QSTR 定义的源文件加入到 Makefile 文件中的变量 SRC_QSTR 中，则在编译期间会自动调用 Python 脚本搜集所有的 QSTR 的关键字，例如，代码 3-11 中的写法。

代码 3-11　在 Makefile 中的 SRC_QSTR 中添加待解析的源文件

```
# List of sources for qstr extraction
SRC_QSTR +=  modmachine.c \
             machine_pin.c \
             machine_adc.c \
             machine_dac.c \
             machine_uart.c \
             machine_spi.c \
             machine_pwm.c \
             machine_timer.c \
             machine_enc.c \
             modutime.c \
             $(BOARD_DIR)/machine_pin_board_pins.c \
```

3.1.2　从 Makefile 追溯编译过程

前面为了验证安装的编译工具链能够正常工作，已经编译过了 minimal 工程。本节将进一步研究 MicroPython 自带 Makefile 文件定义的编译 minimal 工程的具体操作步骤。为了观察 minimal 工程的编译过程，再次运行“make CROSS=1”命令，得到 build log 输出，见代码 3-12。

代码 3-12　编译 minimal 工程的详细步骤

```
$ make CROSS = 1
Use make V = 1 or set BUILD_VERBOSE in your environment to increase build verbosity
mkdir -p build/genhdr
GEN build/genhdr/mpversion.h
GEN build/genhdr/moduledefs.h
GEN build/genhdr/qstr.i.last
GEN build/genhdr/qstr.split
GEN build/genhdr/qstrdefs.collected.h
QSTR updated
GEN build/genhdr/qstrdefs.generated.h
GEN build/genhdr/compressed.split
GEN build/genhdr/compressed.collected
Compressed data updated
GEN build/genhdr/compressed.data.h
mkdir -p build/build/
mkdir -p build/lib/libc/
mkdir -p build/lib/mp-readline/
mkdir -p build/lib/utils/
mkdir -p build/py/
CC ../../py/mpstate.c
CC ../../py/nlr.c
...
CC ../../py/repl.c
CC ../../py/smallint.c
CC ../../py/frozenmod.c
CC main.c
CC uart_core.c
CC ../../lib/utils/printf.c
CC ../../lib/utils/stdout_helpers.c
CC ../../lib/utils/pyexec.c
CC ../../lib/mp-readline/readline.c
MISC freezing bytecode
CC build/_frozen_mpy.c
CC ../../lib/libc/string0.c
LINK build/firmware.elf
   text    data     bss     dec     hex filename
  67296       0    2528   69824   110c0 build/firmware.elf
Create build/firmware.dfu
```

从编译过程输出的信息中可以看到，在真正的编译过程开始之前：

- 先创建了一些头文件，这些头文件大部分都与 QSTR 有关。
- 然后创建了一些目录，似乎是原地创建了一个 build 目录，然后在这个 build 目录下创建了同源代码相同的文件组织结构。

之后，启动 C 编译器开始编译：

- 先编译 py 目录下的 Python 内核源文件。
- 再编译 minimal 项目目录中的 C 源文件。

- 然后编译了引用 lib 目录中的一些源文件。
- 中间有一个生成 frozen 字节码的操作，似乎是在新创建的 build 目录下面生成了一个_frozen_mpy.c 源文件，然后进行了编译。
- 最后通过链接过程，在 build 目录下生成了 firmware.elf，又进一步生成了 firmware.dfu 文件。

在上述过程中，生成 QSTR 相关的头文件和生成_frozen_mpy.c 源文件的操作显然不是 C 编译器能够完成的，由此可以想到是 tools 和 py 目录下的那些 Python 脚本源文件。但具体是通过哪些脚本程序创建的这些文件？并且它们是做什么工作的呢？为此，可根据编译输出信息中的提示，使用“V=1”参数重新编译工程，以获得更详细的编译信息，见代码 3-13。

代码 3-13　minimal 项目输出详细的编译信息(a)

```
$ make CROSS=1 V=1
mkdir -p build/genhdr
python3 ../../py/makeversionhdr.py build/genhdr/mpversion.h
GEN build/genhdr/mpversion.h
```

由代码 3-13 可知，通过执行 py/makeversionhdr.py 脚本程序，将执行结果输出到 build/genhdr/mpversion.h 文件中。看一下生成的 mpversion.h 文件中的内容，见代码 3-14。

代码 3-14　mpversion.h 文件中的 MicroPython 版本信息

```
// This file was generated by py/makeversionhdr.py
#define MICROPY_GIT_TAG "v1.16"
#define MICROPY_GIT_HASH "<no hash>"
#define MICROPY_BUILD_DATE "2021-12-21"
```

从 mpversion.h 文件的注释可以看到，这个文件由 makeversionhdr.py 脚本文件生成，用于从 MicroPython 的代码仓库中提取版本信息。其中，MICROPY_GIT_HASH 的值原本是从.git 目录中提取出来的，但是，作者在制作精简版的代码包时，把.git 目录精简掉了，因此这里没有获得任何有效的 git hash 值信息，指定定义为“<no hash>”。

在详细的编译信息中继续向下看，这里将要创建生成 moduledefs.h 文件，见代码 3-15。

代码 3-15　minimal 项目输出详细的编译信息(b)

```
GEN build/genhdr/moduledefs.h
python3 ../../py/makemoduledefs.py --vpath="., ../.., "  py/mpstate.c ... py/frozenmod.c
extmod/moduasyncio.c ... extmod/uos_dupterm.c lib/embed/abort_.c lib/utils/printf.c build/
genhdr/moduledefs.h > build/genhdr/moduledefs.h
```

通过执行 py/makemoduledefs.py 脚本，传入整个项目中所有的源文件的清单，将输出的信息保存到 build/genhdr/moduledefs.h 中。看一下 moduledefs.h 中文件的内容，见代码 3-16。

代码 3-16　minimal 项目生成的 moduledefs.h 源文件

```
// Automatically generated by makemoduledefs.py.
#if (MICROPY_PY_ARRAY)
    extern const struct _mp_obj_module_t mp_module_uarray;
    #define MODULE_DEF_MP_QSTR_UARRAY { MP_ROM_QSTR(MP_QSTR_uarray), MP_ROM_PTR(&mp_
module_uarray) },
#else
    #define MODULE_DEF_MP_QSTR_UARRAY
#endif
#define MICROPY_REGISTERED_MODULES \
    MODULE_DEF_MP_QSTR_UARRAY \
// MICROPY_REGISTERED_MODULES
```

从 moduledefs.h 源文件中的注释可以看到，该文件明确声明是由 makemoduledefs.py 文件产生的。从文件内容来看，是从 MicroPython 代码仓库中搜集到的某些特定方式定义启用的模块的宏开关。这些生成的宏开关将参与到最终编译 MicroPython 源码的过程中。

在详细的编译信息中继续向下看，这里将要创建生成 qstr.i.last 文件，见代码 3-17。

代码 3-17　minimal 项目输出详细的编译信息(c)

```
GEN build/genhdr/qstr.i.last
python3 ../../py/makeqstrdefs.py pp arm-none-eabi-gcc -E output build/genhdr/qstr.i.last
cflags -I. -I../.. -Ibuild -Wall -Werror -std=c99 -nostdlib -mthumb -mtune=cortex-
m4 -mcpu=cortex-m4 -msoft-float -fsingle-precision-constant -Wdouble-promotion-
Wfloat-conversion -Os -DNDEBUG -fdata-sections -ffunction-sections -DMICROPY_ROM_
TEXT_COMPRESSION=1 -DNO_QSTR cxxflags -DNO_QSTR sources ../../py/mpstate.c ... ../../lib/
utils/printf.c build/genhdr/moduledefs.h ../../py/mpconfig.h mpconfigport.h
```

由“arm-none-eabi-gcc -E”命令对整个项目中所有的 C 源代码进行预编译，展开 #include 的内容并根据 #deifne 的宏开关选出有效的代码，得到“原汁原味”的一整段 C 源程序代码。紧接着，以此输出作为参数，经由 py/makeqstrdefs.py 脚本程序，对编译器预处理输出的内容进行格式化，之后将处理结果文本输出到 build/genhdr/qstr.i.last 文件中。

简单看一下 build/genhdr/qstr.i.last 文件中的代码片段，见代码 3-18。

代码 3-18　minimal 项目生成 qstr.i.last 文件中的代码

```
# 14 "c:\\msys64\\usr\\gcc-arm-none-eabi-10-2020-q4-major\\arm-none-eabi\\
include\\stdint.h" 2 3 4
# 1 "c:\\msys64\\usr\\gcc-arm-none-eabi-10-2020-q4-major\\arm-none-eabi\\
include\\sys\\_stdint.h" 1 3 4
# 20 "c:\\msys64\\usr\\gcc-arm-none-eabi-10-2020-q4-major\\arm-none-eabi\\
include\\sys\\_stdint.h" 3 4
typedef __int8_t int8_t ;
```

这里表示定义 int8_t 的内容重复出现了多次，但最终送给 C 编译器的代码中只有一份。build/genhdr/qstr.i.last 文件更重要的意义在于展开所有的头文件，然后通过 py/makeqstrdefs.py 脚本提取其中类模块、类方法、类属性的 QSTR 字符串，而这些 QSTR 字

符串将在最终的用户 Python 脚本中作为关键字表示具体的对象。

在详细的编译信息中继续向下看，紧接着就要使用 makeqstrdefs.py 脚本处理刚刚生成的 qstr.i.last 文件，将处理结果输出到 qstr 目录下，见代码 3-19。

代码 3-19　minimal 项目输出详细的编译信息(d)

```
GEN build/genhdr/qstr.split
python3 ../../py/makeqstrdefs.py split qstr build/genhdr/qstr.i.last build/genhdr/qstr _
touch build/genhdr/qstr.split
```

此时，从 build/genhdr/qstr.i.last 源文件中抽取的 QSTR 字符串将根据各自的归属关系，以每个文件表示各自的类模块，将类属性和方法提取出来并归类，存放在\build\genhdr\qstr 目录下。在 minimal 项目中，由于暂未加入任何外部的类模块，所以目前能看到的都是 Python 内置的类，如图 3-1 所示。

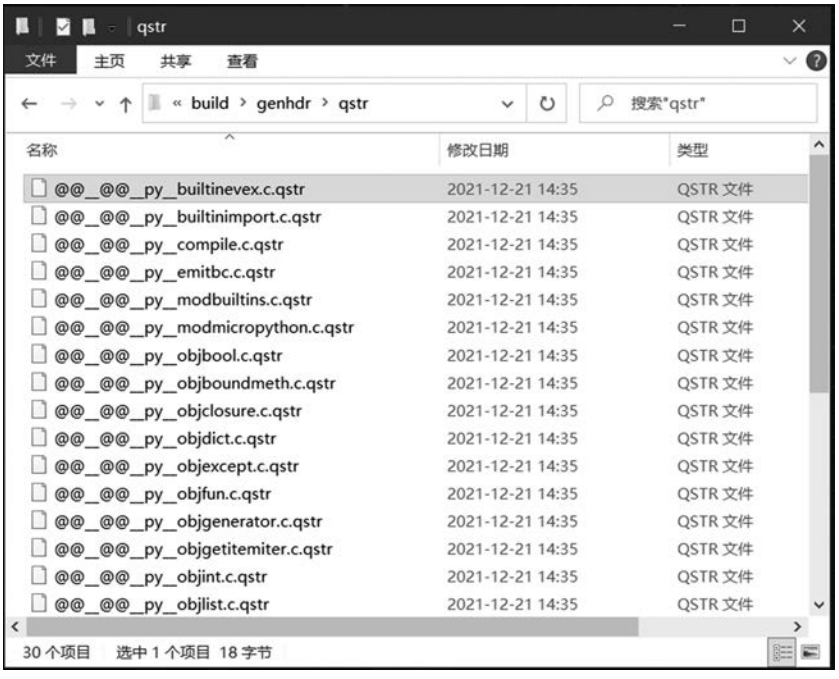

图 3-1　QSTR 字符串对象文件清单

以其中的@@_@@_py_objlist.c.qstr 文件为例，其内部包含了对象 list 类方法的 QSTR 清单，见代码 3-20。

代码 3-20　对象 list 类方法的 QSTR 清单

```
Q(key)
Q(reverse)
Q(append)
Q(clear)
Q(copy)
```

```
Q(count)
Q(extend)
Q(index)
Q(insert)
Q(pop)
Q(remove)
Q(reverse)
Q(sort)
Q(list)
```

在详细的编译信息中继续向下看，紧接着还要进一步将它们整合到一份头文件中，以便于编译器处理，见代码 3-21。

代码 3-21　minimal 项目输出详细的编译信息(e)

```
GEN build/genhdr/qstrdefs.collected.h
python3 ../../py/makeqstrdefs.py cat qstr _ build/genhdr/qstr build/genhdr/qstrdefs.
collected.h
QSTR updated
```

这里通过/py/makeqstrdefs.py 脚本生成的 qstrdefs.collected.h 文件包含了整个工程所有的 QSTR 字符串，但尚未处理重复出现的情况，见代码 3-22。

代码 3-22　qstrdefs.collected.h 中汇总的 QSTR 字符串清单

```
Q(ArithmeticError)

Q(ArithmeticError)

Q(AssertionError)

Q(AssertionError)

Q(AssertionError)

Q(AttributeError)

Q(AttributeError)

Q(BaseException)

Q(BaseException)

...
```

在详细的编译信息中继续向下看，继续处理 QSTR，生成 qstrdefs.generated.h 文件，见代码 3-23。

代码 3-23　minimal 项目输出详细的编译信息(f)

```
GEN build/genhdr/qstrdefs.generated.h
cat ../../py/qstrdefs.h qstrdefsport.h build/genhdr/qstrdefs.collected.h | sed 's/^Q(.*)/
"&"/' | arm-none-eabi-gcc -E -I. -I../.. -Ibuild -Wall -Werror -std=c99 -nostdlib -
mthumb -mtune=cortex-m4 -mcpu=cortex-m4 -msoft-float -fsingle-precision -
constant -Wdouble-promotion -Wfloat-conversion -Os -DNDEBUG -fdata-sections -
ffunction-sections -DMICROPY_ROM_TEXT_COMPRESSION=1 - | sed 's/^\"\(Q(.*)\)\"/\1/'>
build/genhdr/qstrdefs.preprocessed.h
python3 ../../py/makeqstrdata.py build/genhdr/qstrdefs.preprocessed.h > build/genhdr/
qstrdefs.generated.h
```

这里整合了之前生成的 build/genhdr/qstrdefs.collected.h、py/qstrdefs.h 以及 minimal 项目目录下开发者自定义的 qstrdefsport.h 文件文本作为输入，通过 sed 命令进行文本匹配，将其中包含 QSTR 定义的代码语句进一步提取出来，输出到 genhdr/qstrdefs.preprocessed.h 文件。然后，通过 py/makeqstrdata.py 脚本文件，进一步提取 QSTR，并为每一条 QSTR 生成 hash 值，形成键值对，输出到 qstrdefs.generated.h。此时，qstrdefs.generated.h 文件就包含了整个工程中所有对象的 QSTR 字符串名字对应的 hash 表，见代码 3-24。

代码 3-24　minimal 项目中生成的 qstrdefs.generated.h 源文件

```
// This file was automatically generated by makeqstrdata.py

QDEF(MP_QSTRnull, (const byte*)"\x00\x00" "")
QDEF(MP_QSTR_, (const byte*)"\x05\x00" "")
QDEF(MP_QSTR___dir__, (const byte*)"\x7a\x07" "__dir__")
QDEF(MP_QSTR__0x0a_, (const byte*)"\xaf\x01" "\x0a")
QDEF(MP_QSTR__space_, (const byte*)"\x85\x01" " ")
QDEF(MP_QSTR__star_, (const byte*)"\x8f\x01" "*")
QDEF(MP_QSTR__slash_, (const byte*)"\x8a\x01" "/")
QDEF(MP_QSTR__lt_module_gt_, (const byte*)"\xbd\x08" "<module>")
QDEF(MP_QSTR__, (const byte*)"\xfa\x01" "_")
QDEF(MP_QSTR___call__, (const byte*)"\xa7\x08" "__call__")
QDEF(MP_QSTR___class__, (const byte*)"\x2b\x09" "__class__")
QDEF(MP_QSTR___delitem__, (const byte*)"\xfd\x0b" "__delitem__")
QDEF(MP_QSTR___enter__, (const byte*)"\x6d\x09" "__enter__")
QDEF(MP_QSTR___exit__, (const byte*)"\x45\x08" "__exit__")
QDEF(MP_QSTR___getattr__, (const byte*)"\x40\x0b" "__getattr__")
QDEF(MP_QSTR___getitem__, (const byte*)"\x26\x0b" "__getitem__")
QDEF(MP_QSTR___hash__, (const byte*)"\xf7\x08" "__hash__")
QDEF(MP_QSTR___init__, (const byte*)"\x5f\x08" "__init__")
...
```

这些 hash 值将在 MicroPython 运行时被用于索引字符串表示的对象。

之后，MicroPython 的原始开发者还考虑到微控制器平台上存储空间有限，又增加了一个环节，将这些 QSTR 字符串的数据进行压缩存储，见代码 3-25。

代码 3-25　minimal 项目输出详细的编译信息(g)

```
GEN build/genhdr/compressed.split
python3 ../../py/makeqstrdefs.py split compress build/genhdr/qstr.i.last build/genhdr/
compress _touch build/genhdr/compressed.split

GEN build/genhdr/compressed.collected
python3 ../../py/makeqstrdefs.py cat compress _ build/genhdr/compress build/genhdr/
compressed.collected
Compressed data updated

GEN build/genhdr/compressed.data.h
python3 ../../py/makecompresseddata.py build/genhdr/compressed.collected > build/genhdr/
compressed.data.h
```

这里生成的 compressed.data.h 文件会对整个 MicroPython 项目中的字符串数据进行压缩,将复用的字符串单元存入字典 MP_COMPRESSED_DATA,在实际使用的字符串 MP_MATCH_COMPRESSED 中,用索引取代被编入字典的字符串,从而实现数据压缩的功能,见代码 3-26。

代码 3-26　minimal 项目中生成的 compressed.data.h 源文件

```
#define MP_MAX_UNCOMPRESSED_TEXT_LEN (68)
MP_COMPRESSED_DATA("argumen\364can'\364objec\364no\364functio\356mus\364supporte\
344multipl\345assignmen\364keywor\344generato\362ar\347o\346nam\345b\345nonloca\
354require\344wron\347doesn'\364fo\362invali\344missin\347suppor\364t\357typ\
345issubclass(\251empt\371foun\344i\356n\357non-keywor\344rang\345allocatio\
356expressio\356identifie\362tuple/lis\364unexpecte\344\341argument\363hav\345redefine\
344wit\350instanc\345negativ\345sequenc\345conver\364defaul\364expect\363failed\254ha\
363indice\363in\364ou\364outsid\345impor\364inden\364lengt\350memor\371numbe\362synta\
370value\363*\370afte\362a\356byte\363clas\363inde\370isn'\364oute\362tupl\345valu\345'
%q\247fro\355ite\355lis\364lon\347sel\346zer\357\262ba\344a\363i\363o\362\261#%\
344%\361%\365'break'/'continue\247'except\247'yield\247**\370*/*\252\2633-ar\3473\
266<\275>\275BaseExceptio\356GeneratorExi\364LH\323Non\345StopIteratio\356__init__
(\251abort(\251acceptabl\345activ\345allocatin\347alread\371an\344an\371assig\
356attribut\345attribute\363bas\345base\363befor\345bindin\347buffe\362buil\344b\
371callabl\345calle\344characte\362chr(\251classe\363cod\345conflic\364consisten\364")
MP_MATCH_COMPRESSED("'break'/'continue' outside loop", "\377\327\265loop")
MP_MATCH_COMPRESSED("'yield' outside function", "\377\331\265\204")
MP_MATCH_COMPRESSED("* x must be assignment target", "\377\275\205\216\210target")

...

MP_MATCH_COMPRESSED("wrong number of values to unpack", "\377\221\272\214\274\227unpack")
MP_MATCH_COMPRESSED("zero step", "\377\315step")
// Total input length:          2865
// Total compressed length:     1221
// Total data length:           1128
// Predicted saving:            516
```

```
// gzip length:              1755
// Percentage of gzip:       133.8 %
// zlib length:              1743
// Percentage of zlib:       134.8 %
```

从此处的记录信息可以看到，总共 2865 个字节的字符串，经过压缩后，在用 MicroPython 内部的两种不同的压缩组件保存时：gzip 对应长度为 1755 字节；zlib 对应长度为 1743 字节。

接下来的操作就是使用 mkdir 命令，在 build 目录下创建同源代码结构相同的目录结构，用于存放 C 编译器编译生成的 o 文件。例如，在此处截取的编译过程信息的片段中，调用 arm-none-eabi-gcc 命令，将../../py/mpstate.c 文件编译成 build/py/mpstate.o 文件，见代码 3-27。在作者看来，这种设计也体现了作者的“洁癖”，除了原有源代码，所有新生成的文件全部放在 build 目录下，当需要删除编译结果时，也只需要删除整个 build 目录。

代码 3-27　minimal 项目输出详细的编译信息(h)

```
mkdir -p build/build/
mkdir -p build/lib/libc/
mkdir -p build/lib/mp-readline/
mkdir -p build/lib/utils/
mkdir -p build/py/
CC ../../py/mpstate.c
arm-none-eabi-gcc -I. -I../.. -Ibuild -Wall -Werror -std=c99 -nostdlib -mthumb -
mtune=cortex-m4 -mcpu=cortex-m4 -msoft-float -fsingle-precision-constant -
Wdouble-promotion -Wfloat-conversion -Os -DNDEBUG -fdata-sections -ffunction-
sections -DMICROPY_ROM_TEXT_COMPRESSION=1 -c -MD -o build/py/mpstate.o ../../py/
mpstate.c
...
```

在经历了漫长的编译过程之后，最后调用 arm-none-eabi-ld 命令执行链接过程，生成微控制器上的可执行文件 build/firmware.elf，见代码 3-28。

代码 3-28　minimal 项目输出详细的编译信息(i)

```
LINK build/firmware.elf
arm-none-eabi-ld -nostdlib -T stm32f405.ld -Map=build/firmware.elf.map --cref --
gc-sections -o build/firmware.elf build/py/mpstate.o build/py/nlr.o build/py/nlrx86.o
build/py/nlrx64.o build/py/nlrthumb.o ... build/build/_frozen_mpy.o build/lib/libc/string0.o
```

调用 arm-none-eabi-size 命令查看可执行文件中的内存使用情况，见代码 3-29。

代码 3-29　minimal 项目输出详细的编译信息(j)

```
arm-none-eabi-size build/firmware.elf
   text    data     bss     dec     hex filename
  67296       0    2528   69824   110c0 build/firmware.elf
```

调用 arm-none-eabi-objcopy 命令，将 elf 文件转换成二进制的 bin 文件，见代码 3-30。

代码 3-30　minimal 项目输出详细的编译信息(k)

```
arm - none - eabi - objcopy - O binary - j .isr_vector - j .text - j .data build/firmware.elf
build/firmware.bin
```

最后，调用 tools/dfu.py 脚本程序，将 bin 文件打包成 dfu 文件，见代码 3-31。

代码 3-31　minimal 项目输出详细的编译信息(l)

```
Create build/firmware.dfu
python3 ../../tools/dfu.py - b 0x08000000:build/firmware.bin build/firmware.dfu
```

armgcc 编译工具链默认生成的可执行文件是 elf，至于生成 bin 文件乃至 dfu 文件的操作，可根据实际需要考虑是否执行，并相应在 Makefile 中调整编译脚本。

3.2　基于 MM32F3 微控制器移植 minimal 工程

基于对原生 minimal 工程的源代码内容的了解，复制一份 minimal 工程目录，改名为 mm32f3，再对应地将于微控制器型号和电路板相关的具体实现代码从默认的 STM32 微控制器平台替换到 MM32F3 微控制器平台。本节将说明在 MM32F3 微控制器上进行移植的具体过程。

3.2.1　在 lib 目录中添加 MindSDK 代码

根据 MicroPython 整体的源文件组织结构，第三方的库、代码包均放置于 lib 目录下，在完整版的 MicroPython 代码包中可以看到，stm32lib、nxp_driver 等微控制器固件开发包也在 lib 目录下。因此，可在 lib 目录下也创建了 mm32mcu 的目录，用于存放 MindSDK 的源文件，用于实现 MicroPython 工程向 MM32F3 微控制器底层硬件的访问。在 MicroPython 中添加 MindSDK 驱动代码，如图 3-2 所示。

其中包含了 MM32F3270 微控制器的芯片头文件、为 armgcc 编译器准备的启动程序和链接命令文件，以及片上外设模块驱动程序。

这里，在 mm32mcu 目录下再创建次级目录 mm32f3270，是考虑到后续还有机会支持 mm32mcu 的其他系列微控制器，以后可以在与 mm32f3270 并列的级别上再创建 mm32xxxx 目录即可。

3.2.2　在 ports 目录中创建 mm32f3 项目目录

另外，需要在 ports/mm32f3 目录下建立与电路板相关的目录 boards，之后再在下级创建具体的板子目录 plus-f3270。这里考虑的是，同样使用 mm32f3270 的微控制器，可以做出多种不同的开发板，涉及其中的引脚功能分配、时钟配置等，因此单独为每一块电路板创建独立的目录。与硬件电路无关的通用功能的实现代码，将放在 mm32f3 根目录下。此时，mm32f3 的目录结构如图 3-3 所示。

图 3-2　向 MicroPython 添加 MindSDK 源码

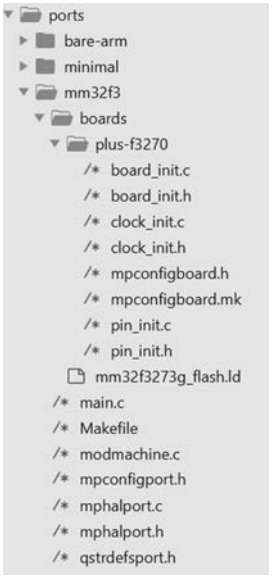

图 3-3　新建 mm32f3 移植项目目录结构

需要特别注意的是，相对于基于 STM32F4 微控制器创建的 minimal 项目，新建基于 MM32F3 微控制器的 mm32f3 项目的源文件组织结构进行了微调：

- 删除了与 stm32 平台相关的 stm32f405.ld 文件。在 boards 目录下，复制了来自 mm32mcu 目录的 mm32f3273g_flash.ld 文件，并进行了微调。
- 删除了 uart_core.c，与 REPL 串口相关的接口函数的实现被收纳到新创建 mphalport.c 文件中。
- 删除了 frozentest.mpy 和 frozentest.py 文件。在 mm32f3 项目中，暂时不考虑使用预编译 Python 模块。
- 增加了 modmachine.c 文件，在其中创建了一个简单的 machine 模块的实现，作为后续硬件相关模块的容器。
- 精简了 main.c 文件。将原来在 main.c 文件中定义的硬件中断向量表的部分移除，直接使用来自 MindSDK 启动代码中的中断向量表定义。移除了实际没有执行的条件编译分支。

实际上，这些文件结构的调整，借鉴了同在 ports 目录下的其他正式移植项目的文件组织结构。基于 minimal 工程，再结合正式移植项目的文件组织结构，便形成了适合正式开发的最小工程。

3.2.2.1　新建 boards 目录

在 boards 目录下细分具体板的子目录，例如，此处为 PLUS-F3270 板新建的目录，其中存放来自 MindSDK 的样例工程（如 hello_world 工程）中的关于配置板相关的源文件：

board_init.h/.c、clock_init.h/.c、pin_init.h/c。顾名思义，board_init.h/.c 中定义了 BOARD_Init()函数，这个函数将在 main()函数的最开始被调用，用于初始化开发板的硬件电路。BOARD_Init()函数内部调用的时钟系统初始化和微控制器芯片引脚复用功能的函数分别在 clock_init.h/c 和 pin_init.h/.c 文件中实现。

在 mm32f3 的最小移植工程中，通过 BOARD_Init()函数至少实现两个功能：

- 配置当前系统内核主频为 120MHz。
- 启用 UART1，并配置其波特率为 115 200bps。

后续将要使用的新的外设，如果不需要在运行时动态开关时钟或者配置引脚复用功能，则需要在此处添加配置时钟和引脚复用功能的源代码。

mpconfigboard.mk 文件是供 Makefile 文件引用的，其中指定同本开发板及板载微控制器的相关定义，见代码 3-32。

代码 3-32　mm32f3 项目的 mpconfigboard.mk 源文件

```
MCU_SERIES = mm32f3270
CMSIS_MCU = mm32f3273g
LD_FILES = boards/mm32f3273g_flash.ld
```

mpconfigboard.h 文件也定义了板子的名称和板载微控制器芯片的具体型号，这里指定的字符串将在 MicroPython 运行时从 REPL 输出，见代码 3-33。

代码 3-33　mm32f3 项目的 mpconfigboard.h 源文件

```
#ifndef __MPCONFIGBOARD_H__
#define __MPCONFIGBOARD_H__

#define MICROPY_HW_BOARD_NAME "PLUS-F3270"
#define MICROPY_HW_MCU_NAME   "MM32F3273G9P"

#endif /* __MPCONFIGBOARD_H__ */
```

3.2.2.2　调整链接命令文件 mm32f3273g_flash.ld

在微调 mm32f3273g_flash.ld 文件时，可将 MM32F3270 微控制器片内集成的 128KB 内存分成两个 64KB 的部分，并将后半部分单独分配给 MicroPython 的 GC 存储管理器使用，前半部分给整个 MM32F3270 程序的系统堆和栈使用，见代码 3-34。

代码 3-34　mm32f3 项目的链接命令文件(a)

```
/* Specify the memory areas */
MEMORY
{
  m_interrupts          (RX)  : ORIGIN = 0x08000000, LENGTH = 0x00000400
  m_text                (RX)  : ORIGIN = 0x08000400, LENGTH = 0x0007FC00 /* 512KB. */
  m_data                (RW)  : ORIGIN = 0x20000000, LENGTH = 0x00010000 /* 64KB.  */
  m_data2               (RW)  : ORIGIN = 0x20010000, LENGTH = 0x00010000 /* 64KB.  */
/* make a standalone memory block for micropython gc. */
}
```

然后，在 mm32f3273g_flash.ld 文件中创建一些关于堆和栈地址和长度的变量，用于即将微调的 main.c 的初始化 MicroPython 的过程，见代码 3-35。

代码 3-35　mm32f3 项目的链接命令文件(b)

```
_estack = __StackTop;
_sstack = __StackLimit;

/* Do not use the traditional C heap. */
__heap_size__ = 0;

/* use micropython gc. */
_gc_heap_start = ORIGIN(m_data2);
_gc_heap_end = ORIGIN(m_data2) + LENGTH(m_data2);
```

关于完整的 mm32f3273g_flash.ld 文件内容，可在本书配套资源的代码包中查阅。

3.2.2.3　新建 mphalport.c 文件

在原版的 minimal 工程中，只有 mphalport.h 文件，没有 mphalport.c 文件。对应于在正式的移植项目中使用 mphalport.h 定义 MicroPython 内核操作微控制器平台引脚的宏函数，mm32f3 项目中创建了 mphalport.c 文件，收纳了 uart_core.c 文件中的内容，用于定义 MicroPython 内核操作 UART 的函数。MicroPython 内核操作 UART，主要是实现 REPL。新创建 mphalport.c 文件中的内容，见代码 3-36。

代码 3-36　mm32f3 项目中新建 mphalport.c 源文件

```
/* mphalport.c */

#include "py/runtime.h"
#include "py/stream.h"
#include "py/mphal.h"

#include "board_init.h"
#include "hal_uart.h"

int mp_hal_stdin_rx_chr(void)
{
    while ( 0u == (UART_STATUS_RX_DONE & UART_GetStatus(BOARD_DEBUG_UART_PORT)) )
    {}
    return UART_GetData(BOARD_DEBUG_UART_PORT);
}

void mp_hal_stdout_tx_strn(const char *str, mp_uint_t len)
{
    while (len--)
    {
        while ( 0u == (UART_STATUS_TX_EMPTY & UART_GetStatus(BOARD_DEBUG_UART_PORT)) )
        {}
        UART_PutData(BOARD_DEBUG_UART_PORT, *str++);
```

```
    }
}

/* EOF. */
```

mp_hal_stdin_rx_chr()和 mp_hal_stdout_tx_strn()这两个函数的声明位于 MicroPython 内核源文件 py/mphal.h 中。这个文件中还声明了其他的内核操作硬件的函数，大体分为三大类：UART 串口、延时和引脚，见代码 3-37。在后续讲解增加新功能的时候，会逐渐将它们完整实现，当然，实现函数也将存放在 mphalport.c 文件中。但目前，只要实现关于 UART 的两个函数就已经可以先让 MicroPython 运行起来了。

代码 3-37　mphal.h 源文件

```
#ifndef MICROPY_INCLUDED_PY_MPHAL_H
#define MICROPY_INCLUDED_PY_MPHAL_H

#include <stdint.h>
#include "py/mpconfig.h"

#ifdef MICROPY_MPHALPORT_H
#include MICROPY_MPHALPORT_H
#else
#include <mphalport.h>
#endif

/* REPL 终端的相关函数 */
#ifndef mp_hal_stdio_poll
uintptr_t mp_hal_stdio_poll(uintptr_t poll_flags);
#endif

#ifndef mp_hal_stdin_rx_chr
int mp_hal_stdin_rx_chr(void);
#endif

#ifndef mp_hal_stdout_tx_str
void mp_hal_stdout_tx_str(const char *str);
#endif

#ifndef mp_hal_stdout_tx_strn
void mp_hal_stdout_tx_strn(const char *str, size_t len);
#endif

#ifndef mp_hal_stdout_tx_strn_cooked
void mp_hal_stdout_tx_strn_cooked(const char *str, size_t len);
#endif

/* 延时相关的函数 */
#ifndef mp_hal_delay_ms
```

```
void mp_hal_delay_ms(mp_uint_t ms);
#endif

#ifndef mp_hal_delay_us
void mp_hal_delay_us(mp_uint_t us);
#endif

#ifndef mp_hal_ticks_ms
mp_uint_t mp_hal_ticks_ms(void);
#endif

#ifndef mp_hal_ticks_us
mp_uint_t mp_hal_ticks_us(void);
#endif

#ifndef mp_hal_ticks_cpu
mp_uint_t mp_hal_ticks_cpu(void);
#endif

#ifndef mp_hal_time_ns
// Nanoseconds since the Epoch.
uint64_t mp_hal_time_ns(void);
#endif

/* 引脚对象相关的函数 */
// "virtual pin" API from the core.
#ifndef mp_hal_pin_obj_t
#define mp_hal_pin_obj_t mp_obj_t
#define mp_hal_get_pin_obj(pin) (pin)
#define mp_hal_pin_read(pin) mp_virtual_pin_read(pin)
#define mp_hal_pin_write(pin, v) mp_virtual_pin_write(pin, v)
#include "extmod/virtpin.h"
#endif

#endif // MICROPY_INCLUDED_PY_MPHAL_H
```

3.2.2.4　精简 main.c 文件

minimal 项目中的 main.c 文件包含了微控制器平台的中断向量表。在 mm32f3 项目中，可以直接使用 MindSDK 中提供的启动代码中的中断向量表定义，替换掉原有的实现。同时，在具体的 mm32f3 项目中，已经明确仅使用 REPL 进行人机交互，不在代码中解析字符串，也不执行预编译脚本，因此可以进一步简化 main()函数的代码。另外，借鉴 MicroPython 中 mimxrt 项目对栈和堆的初始化操作，在代码里显式调用了 mp_stack_set_top()和 mp_stack_set_limit()函数管理 MicroPython 的栈空间，用 gc_init()管理 MicroPython 的堆空间。实际上，此处对 main()函数的精简操作，在 MicroPython 其他新平台以及后续的版本中，已经逐渐演变成事实上的标准实现。

精简后的 main()函数更加简单直观，见代码 3-38。

代码 3-38　mm32f3 项目中的 main()函数

```
#include "py/compile.h"
#include "py/runtime.h"
#include "py/gc.h"
#include "py/mperrno.h"
#include "py/stackctrl.h"
#include "lib/utils/gchelper.h"
#include "lib/utils/pyexec.h"

#include "board_init.h"

extern uint8_t _sstack, _estack, _gc_heap_start, _gc_heap_end;

int main(void)
{
    /* 初始化电路板相关配置 */
    BOARD_Init();

    /* 初始化 MicroPython 使用的堆栈 */
    mp_stack_set_top(&_estack);
    mp_stack_set_limit(&_estack - &_sstack - 1024);

    for (;;)
    {
        gc_init(&_gc_heap_start, &_gc_heap_end);
        mp_init();
        mp_obj_list_init(MP_OBJ_TO_PTR(mp_sys_path), 0);
        mp_obj_list_append(mp_sys_path, MP_OBJ_NEW_QSTR(MP_QSTR_));
        mp_obj_list_init(MP_OBJ_TO_PTR(mp_sys_argv), 0);

        /* 执行 REPL */
        for (;;) {
            if (pyexec_mode_kind == PYEXEC_MODE_RAW_REPL) {
                if (pyexec_raw_repl() != 0) {
                    break;
                }
            } else {
                if (pyexec_friendly_repl() != 0) {
                    break;
                }
            }
        }

        mp_printf(MP_PYTHON_PRINTER, "MPY: soft reboot\n");
        gc_sweep_all();
        mp_deinit();
    }
}
```

```
...

/* 将微控制器芯片启动代码引导到 main()函数 */
void _start(void)
{
    main();

    for (;;)
    {
    }
}
```

这里要注意，MicroPython 的复位中断服务程序中定义的芯片启动执行过程，在对运行时的内存环境进行了一系列初始化之后，通过_start()函数跳转到应用程序。因此，此处在 main.c 文件中直接定义 main()函数是不能启动过程调用的，需要通过_start()函数进行转接。

商用编译开发环境 Keil MDK 或 IAR 已经将从复位中断服务程序到 main()函数的执行过程封装成运行时库，而 GCC 编译器需要用户通过源代码全部实现这个启动过程。这个过程包括一系列初始化编译器库和电路系统工作环境的操作，具体包括重定向中断向量表、初始化.data 段和.bss 段中的变量、跳转到用户程序等。关于跳转到用户程序，这里跳转到的是_start()函数，而不是开发者更熟悉的 main()函数。这样做的好处是，在进入用户常用的 main()函数之前，一些高级的开发者可能会需要在进入一般意义的 main()之前，留一些可操作的余地，在必要的情况下，可以在 C 语言层面继续执行一些准备工作。这里列举 mm32f3 项目的启动代码，见代码 3-39。

代码 3-39　mm32f3 项目的启动代码

```
Reset_Handler:
    cpsid   i               /* 关总中断 */
    .equ    VTOR, 0xE000ED08
    ldr     r0, = VTOR
    ldr     r1, = __isr_vector
    str     r1, [r0]
    ldr     r2, [r1]
    msr     msp, r2
#ifndef __NO_SYSTEM_INIT
    ldr   r0, = SystemInit
    blx   r0
#endif
/* 搬运部分数据从只读存储区到 RAM 中。搬运数据的区域范围由链接命令文件中的如下符号指定:
- __etext: 代码段的结束位置，数据段的开始位置，也是将要复制的开始位置。
- __data_start __/__ data_end __: 数据段内容应该存放在 RAM 中的位置。
- __noncachedata_start __/__ noncachedata_end __: 不可缓存的内存区，必须要 4 字节对齐。
*/
    ldr     r1, = __etext
    ldr     r2, = __data_start __
```

```
    ldr     r3, = __data_end__

#ifdef __PERFORMANCE_IMPLEMENTATION
/* 这里有另一种遍历存储区的实现:第一种对性能进行优化,第二种对代码量进行优化,模式使用
第二种。可通过定义__PERFORMANCE_IMPLEMENTATION 切换使用第一种方式的实现 */
    subs        r3, r2
    ble         .LC1
.LC0:
    subs        r3, #4
    ldr         r0, [r1, r3]
    str         r0, [r2, r3]
    bgt         .LC0
.LC1:
#else  /* 对代码量进行优化的实现 */
.LC0:
    cmp         r2, r3
    ittt        lt
    ldrlt       r0, [r1], #4
    strlt       r0, [r2], #4
    blt         .LC0
#endif
#ifdef __STARTUP_CLEAR_BSS
/* 人工清零 BSS 段的内容。通常应由 C 库的启动代码实现,如果某个特定编译器的 C 库不支持清
零 BSS 段的操作,需要定义这个宏,启用人工清零 BSS 段。BSS 段的地址范围由链接命令文件中定
义如下符号指定:
- __bss_start__: BSS 段的开始位置,4 字节对齐。
- __bss_end__: BSS 段的结束位置,4 字节对齐。
*/
    ldr r1, = __bss_start__
    ldr r2, = __bss_end__

    movs    r0, 0
.LC5:
    cmp     r1, r2
    itt     lt
    strlt   r0, [r1], #4
    blt     .LC5
#endif      /* __STARTUP_CLEAR_BSS */

    cpsie   i        /* 打开总中断 */
#ifndef __START
#define __START _start
#endif
#ifndef __ATOLLIC__
    ldr     r0, = __START
    blx     r0
#else
    ldr     r0, = __libc_init_array
    blx     r0
```

```
    ldr     r0, = main
    bx      r0
#endif
    .pool
    .size Reset_Handler, . - Reset_Handler
```

从启动代码中还看到一个要点，启动代码中对 BSS 段的初始化并不是默认开启的，需要通过__STARTUP_CLEAR_BSS 宏选项开启。BSS 段中存放的是程序中未赋初值的全局变量，这些全局变量不是默认为 0 的，只有启用了__STARTUP_CLEAR_BSS 选项，才会由启动程序逐一赋值为 0。如果没有人为写 0，那么位于内存中的变量的初值将会是随机的，并且每次都不一样，具体取决于当时 SRAM 中不十分稳定的电路状态。

作者在早期向 MicroPython 中集成 TinyUSB 协议栈时，就在这个 BSS 段的初始化问题上遇到了麻烦。在 USB 的程序中要根据之前协议栈的状态判定如何处理当前事件，这个状态当然是通过一个全局变量实现的，但同样的协议栈代码在 MindSDK 中就是正常工作的，Keil MDK、IAR 和 GCC 都验证可行，但在 MicroPython 的工程里就不能正常工作（连 USB 设备枚举都过不去）。后来经过大量的排查工作，才发现是表示当前状态的全局变量值不对，在启用 USB 协议栈之前，将这些有问题的全局变量人为赋值为 0，程序就能正常工作了。在软件中，协议栈内部的状态应该在协议栈初始化函数中完成初始化，在协议栈之上的应用层粗暴地操作协议栈内部的状态显然是不合理的，但作者又不想轻易地在第三方协议栈内部“动刀子”破坏原有协议栈的完整性，况且同样的代码在另外的编译环境也可以正常工作。最后排查到启动代码这部分，MindSDK 的 armgcc 环境的样例工程和 MicroPython 也是同一份，不一样的就只能是编译器选项了。果然，之前的 MicroPython 的 Makefile 中定义的 CFLAG 中，没有“-D__STARTUP_CLEAR_BSS”，修正之后，最终优雅地解决了问题。

当时这个案例特别难查的一个原因还在于，TinyUSB 的一些全局变量并不是定义在 C 函数外面的，而是在使用的时候才在函数内部用 static 关键字声明其为具有部分全局变量属性的“静态变量”。这也为平时编程提供了一条经验教训：在 C 语言层面，哪怕是部分全局变量属性的“静态变量”，也建议放在函数外面定义，并且应很明确地在定义变量时就赋予明确的初值，或是在整个协议栈的初始化环节，把所有表示状态的变量全部赋予明确的初值。这条经验对于嵌入式系统软件开发尤其宝贵，与桌面软件的集成开发环境中对用户程序重重保护不同，在嵌入式系统中，软件工作的每个环节都是需要明确控制的，尤其是在内存管理方面特别容易出问题。

3.2.2.5　更新 Makefile 文件

相对于 minimal 工程的 Makefile，mm32f3 项目的中将不再包含调用 mpy-cross 编译 frozentest.py 的步骤，移除了生成 DFU 文件的过程，并引入了参数 board，可以用同一个 Makefile 编译多种板子的可执行程序。实际上，在调整 Makefile 时，更多地借鉴了现有的成品移植项目 mimxrt 中 Makefile 的写法。另外，又添加了一些注释说明，让 Makefile 的内容更规整，方便后续开发过程中需要增加模块时，在 Makefile 中明确指定添加的位置。

下面详细解释 mm32f3 项目的 Makefile 的实现内容，以及后续增加新模块时更新 Makefile 的规则，见代码 3-40。

代码 3-40　mm32f3 项目的 Makefile 源文件(a)

```
BOARD ? = plus - f3270
BOARD_DIR ? = boards/ $ (BOARD)
BUILD ? = build - $ (BOARD)

CROSS_COMPILE ? = arm - none - eabi -

ifeq ( $ (wildcard $ (BOARD_DIR)/.),)
$ (error Invalid BOARD specified: $ (BOARD_DIR))
endif

include ../../py/mkenv.mk
include $ (BOARD_DIR)/mpconfigboard.mk

# qstr definitions (must come before including py.mk)
QSTR_DEFS = qstrdefsport.h
QSTR_GLOBAL_DEPENDENCIES = $ (BOARD_DIR)/mpconfigboard.h

# include py core make definitions
include $ (TOP)/py/py.mk
```

在 Makefile 开始的位置,首先解析 BOARD 参数,这个参数同运行 make 命令时使用,见代码 3-41。

代码 3-41　mm32f3 项目的 Makefile 源文件(b)

```
$ make BOARD = plus - f3270
```

这里还包含了一些预先定义的.mk 文件(也是 Makefile 文件,mk 文件相对于 Makefile,类似于.h 文件对于 C 文件),例如,py/mkenv.mk 和 py/py.mk,都是在 MicroPython 内核中定义好的,这里遵循已有项目的用法即可。$(BOARD_DIR)/mpconfigboard.mk 是在 boards 目录中自定义的,其中包含了与定制板子相关的配置,通过 MCU_SERIES 和 CMSIS_MCU 指定的芯片型号,实际会用来组合生成查找芯片启动文件和驱动文件的路径,而 LD_FILES 指定当前这块板子的移植项目所用到的链接命令文件,例如,ports\mm32f3\boards\plus-f3270 目录下的 mpconfigboard.mk 文件,见代码 3-42。

代码 3-42　mm32f3 项目的 mpconfigboard.mk 源文件

```
MCU_SERIES = mm32f3270
CMSIS_MCU = mm32f3273g
LD_FILES = boards/mm32f3273g_flash.ld
```

接下来 Makefile 文件就要定义编译项目的各个选项,其中的内容大体同微控制器开发者熟悉的其他工具链(如 Keil MDK、IAR 等)相似。

指定头文件搜索路径,见代码 3-43。

代码 3-43　mm32f3 项目的 Makefile 源文件(c)

```
MCU_DIR = lib/mm32mcu/$(MCU_SERIES)

# includepath.
INC += -I.
INC += -I$(TOP)
INC += -I$(BUILD)
INC += -I$(BOARD_DIR)
INC += -I$(TOP)/lib/cmsis/inc
INC += -I$(TOP)/$(MCU_DIR)/devices/$(CMSIS_MCU)
INC += -I$(TOP)/$(MCU_DIR)/drivers
```

指定 GCC 编译工具链的编译选项,用 flags 表示,包含 C 编译器选项 CFLAGS 和链接命令选项 LDFLAGS,见代码 3-44。

代码 3-44　mm32f3 项目的 Makefile 源文件(d)

```
# flags.
CFLAGS = $(INC)
CFLAGS +=  -Wall -Werror
CFLAGS += -std=c99
CFLAGS += -nostdlib
CFLAGS += -mthumb
CFLAGS += $(CFLAGS_MCU_$(MCU_SERIES))
CFLAGS += -fsingle-precision-constant -Wdouble-promotion
CFLAGS += -D__STARTUP_CLEAR_BSS

CFLAGS_MCU_CM7  = -mtune=cortex-m7 -mcpu=cortex-m7 -mfloat-abi=hard
                  -mfpu=fpv5-d16
CFLAGS_MCU_CM4  = -mtune=cortex-m4 -mcpu-cortex-m4 -msoft-float
CFLAGS_MCU_CM3  = -mtune=cortex-m3 -mcpu=cortex-m3 -msoft-float
CFLAGS_MCU_CM0P = -mtune=cortex-m0plus -mcpu=cortex-m0plus -msoft-float

ifeq ($(MCU_SERIES), mm32f3270)
CFLAGS += $(CFLAGS_MCU_CM3)
else
$(error Invalid MCU_SERIES specified: $(MCU_SERIES))
endif

CFLAGS += -DMCU_$(MCU_SERIES) -D__$(CMSIS_MCU)__
LDFLAGS = -nostdlib $(addprefix -T,$(LD_FILES)) -Map=$@.map --cref
LIBS = $(shell $(CC) $(CFLAGS) -print-libgcc-file-name)

# Tune for Debugging or Optimization
ifeq ($(DEBUG),1)
CFLAGS += -O0 -ggdb
else
CFLAGS += -Os -DNDEBUG
LDFLAGS += --gc-sections
```

```
CFLAGS += -fdata-sections -ffunction-sections
endif
```

继续向项目中添加源文件，包括 C 源文件清单 SRC_C 和汇编源文件清单 SRC_S、SRC_SS，同时为了方便管理，还用符号 SRC_HAL_MM32_C 和 SRC_BRD_MM32_C 对表示的文件清单分组，见代码 3-45。

代码 3-45　mm32f3 项目的 Makefile 源文件(e)

```
# source files.
SRC_HAL_MM32_C += \
    $(MCU_DIR)/devices/$(CMSIS_MCU)/system_$(CMSIS_MCU).c \
    $(MCU_DIR)/drivers/hal_rcc.c \
    $(MCU_DIR)/drivers/hal_gpio.c \
    $(MCU_DIR)/drivers/hal_uart.c \


SRC_BRD_MM32_C += \
    $(BOARD_DIR)/clock_init.c \
    $(BOARD_DIR)/pin_init.c \
    $(BOARD_DIR)/board_init.c \


SRC_C += \
    main.c \
    modmachine.c \
    mphalport.c \
    lib/libc/string0.c \
    lib/mp-readline/readline.c \
    lib/utils/gchelper_native.c \
    lib/utils/printf.c \
    lib/utils/pyexec.c \
    lib/utils/stdout_helpers.c \
    $(SRC_HAL_MM32_C) \
    $(SRC_BRD_MM32_C) \


# also use cm3 as gchelper_m3.s
ifeq ($(MCU_SERIES),mm32f3270)
SRC_S = lib/utils/gchelper_m3.s
else
SRC_S = lib/utils/gchelper_m0.s
endif

SRC_SS = $(MCU_DIR)/devices/$(CMSIS_MCU)/startup_$(CMSIS_MCU).S
```

特别需要注意的是，新增的类模块若需要使用 QSTR 字符串指示类模块以及类属性和类方法，但不想手动向 qstrdefport.h 文件中添加 QSTR 关键字，则需要将包含新创建类模块定义的源文件加到 SRC_QSTR 文件清单中，交由编译过程中的脚本自动提取 QSTR，见

代码 3-46。

代码 3-46　mm32f3 项目的 Makefile 源文件(f)

```
# list of sources for qstr extraction
SRC_QSTR += modmachine.c
```

指定 obj 文件的存放位置。由源代码文件编译生成的 obj 文件,在 build 目录下以同样的目录结构存放,见代码 3-47。

代码 3-47　mm32f3 项目的 Makefile 源文件(g)

```
# output obj file.
OBJ += $(PY_O)
OBJ += $(addprefix $(BUILD)/, $(SRC_C:.c=.o))
OBJ += $(addprefix $(BUILD)/, $(SRC_S:.s=.o))
OBJ += $(addprefix $(BUILD)/, $(SRC_SS:.S=.o))
```

指定可执行文件的生成规则,见代码 3-48。

代码 3-48　mm32f3 项目的 Makefile 源文件(h)

```
# rules.
all: $(BUILD)/firmware.bin

$(BUILD)/firmware.elf: $(OBJ)
    $(ECHO) "LINK $@"
    $(Q)$(LD) $(LDFLAGS) -o $@ $^ $(LIBS)
    $(Q)$(SIZE) $@

$(BUILD)/firmware.bin: $(BUILD)/firmware.elf
    $(Q)$(OBJCOPY) -O binary $^ $@

$(BUILD)/firmware.hex: $(BUILD)/firmware.elf
    $(Q)$(OBJCOPY) -O ihex -R .eeprom $< $@
```

在代码 3-48 中,可以指定最后生成可执行文件的类型,默认是 elf 文件,也可以根据需要再将 elf 文件转成 bin 或 hex 文件。

最后,按照 MicroPython 所有移植项目 Makefile 都遵循的做法,还要再包含一个 mk 文件,引用 MicroPython 定义的通用的工程项目规则,见代码 3-49。

代码 3-49　mm32f3 项目的 Makefile 源文件(i)

```
include $(TOP)/py/mkrules.mk
```

在后续的开发过程中,关于 Makefile 文件,应重点关注与常用操作相关的几个地方:

- 在 INC 中添加新的源文件搜索路径。
- 在 SRC_C 中添加新的源文件,例如,更多的微控制器底层驱动程序源文件、实现新的类模块的源文件等。

- 在 SRC_QSTR 中添加新的实现新的类模块的源文件，给编译过程自动解析 QSTR 关键字。

至此，基于 MM32F3 微控制器的 MicroPython 最小工程 mm32f3 项目就移植完毕了，准备好了源代码和 Makefile，在命令行窗口中进入 ports/mm32f3 目录下，运行 make 命令开始编译，见代码 3-50。

代码 3-50　编译 mm32f3 项目

```
Andrew@Andrew - PC MSYS /c/_git_repo/micropython_su/micropython - 1.16 - mini/ports/mm32f3_v0.1
# make BOARD = plus - f3270
Use make V = 1 or set BUILD_VERBOSE in your environment to increase build verbosity
mkdir -p build - plus - f3270/genhdr
GEN build - plus - f3270/genhdr/mpversion.h
GEN build - plus - f3270/genhdr/moduledefs.h
GEN build - plus - f3270/genhdr/qstr.i.last
GEN build - plus - f3270/genhdr/qstr.split
GEN build - plus - f3270/genhdr/qstrdefs.collected.h
QSTR updated
GEN build - plus - f3270/genhdr/qstrdefs.generated.h
mkdir -p build - plus - f3270/boards/plus - f3270/
mkdir -p build - plus - f3270/extmod/
mkdir -p build - plus - f3270/lib/embed/
mkdir -p build - plus - f3270/lib/libc/
mkdir -p build - plus - f3270/lib/mm32mcu/mm32f3270/devices/mm32f3273g/
mkdir -p build - plus - f3270/lib/mm32mcu/mm32f3270/drivers/
mkdir -p build - plus - f3270/lib/mp - readline/
mkdir -p build - plus - f3270/lib/utils/
mkdir -p build - plus - f3270/py/
CC ../../py/mpstate.c
CC ../../py/nlr.c
...
CC ../../lib/utils/printf.c
CC main.c
CC modmachine.c
CC mphalport.c
CC ../../lib/libc/string0.c
CC ../../lib/mp - readline/readline.c
CC ../../lib/utils/gchelper_native.c
CC ../../lib/utils/pyexec.c
CC ../../lib/utils/stdout_helpers.c
CC ../../lib/mm32mcu/mm32f3270/devices/mm32f3273g/system_mm32f3273g.c
CC ../../lib/mm32mcu/mm32f3270/drivers/hal_rcc.c
CC ../../lib/mm32mcu/mm32f3270/drivers/hal_gpio.c
CC ../../lib/mm32mcu/mm32f3270/drivers/hal_uart.c
CC boards/plus - f3270/clock_init.c
CC boards/plus - f3270/pin_init.c
CC boards/plus - f3270/board_init.c
AS ../../lib/utils/gchelper_m3.s
CC ../../lib/mm32mcu/mm32f3270/devices/mm32f3273g/startup_mm32f3273g.S
```

```
LINK build-plus-f3270/firmware.elf
   text    data     bss     dec     hex filename
  87396       0    2416   89812   15ed4 build-plus-f3270/firmware.elf
```

编译通过，生成可执行文件 build-plus-f3270/firmware.elf。

3.3　首次在 MM32F3 微控制器上运行 MicroPython

3.3.1　下载可执行文件到 MM32F3 微控制器

使用 Keil MDK 或者 IAR 等使用图形界面的开发环境，可以在图形界面环境下编译源码工程，并将编译生成的可执行文件下载到目标微控制器中。但若使用 ARMGCC 等命令行工具链，则需要额外的下载工具，才能将编译生成的可执行文件下载到目标微控制器中。

若使用 SEGGER J-Link 调试器，可以搭配 SEGGER Ozone 软件或者 J-FLASH、J-FLASH Lite 实现单独下载的功能。但 J-Link 调试器价格昂贵，并且老版本的 J-LINK 调试器无法支持更新的微控制设备。相比而言，开源的 DAP-Link 方案更接地气。可以适配 DAP-Link 的命令行工具有 openocd、pyocd 等，但这些工具对某些具体微控制器设备的兼容性并不是很好，时不时会出现不识别设备或者连接不上的情况，需要开发者自行调试才能确保它们能够正常工作。同时，基于命令的操作方式，对于已经习惯了在图形界面环境下调试的开发者而言，也不是很友好。不过，如果需要搭建持续集成和自动化测试系统，那么这些基于命令行的工具仍是不可或缺的。

对于仅专注于微控制器端的软件开发者而言，希望能够以最简单的方式解决单独下载可执行文件的问题。本节总结了几种简单易用的方法，专门针对使用 DAP-Link 调试器的情况，通过常用图形界面工具，实现单独下载可执行文件的功能，从而将 MicroPython 固件文件下载到 MM32F3 微控制器。

3.3.1.1　借用 Keil 工程

Keil IDE 实现下载功能的部分，相对于编译过程，在内部应该也是一个独立的小工具，这是可以实现用 Keil 下载程序的关键。Keil 没有将内部的下载工具独立确认，因此，还需要创建一个不包含任何源码的空工程，跳过编译部分，仅使用其中下载程序的功能。具体操作步骤如下：

(1) 启动 Keil IDE，创建新工程，并选定设备类型为目标微控制器设备。

例如，在本机的“D:_worksapce\keil\mm32f3”目录下，创建了 mm32f3.uvprojx 工程文件，见图 3-4。

(2) 在 Output 选项卡中，指定将要下载的可执行文件的路径。

在样例中，mm32f3 目录下存放了 micropython.hex 文件。单击 Select Folder for Objects 按钮，指定为 mm32f3 目录，然后在 Name of Executable 对应的文本框中输入文件名 micropython.hex，如图 3-5 所示。

从字面上看，这里指定的是编译输出的路径和文件名，但实际上，下载过程同编译过程是绑定的，编译过程生成的可执行文件，将被 Keil 自动作为下载过程的输入文件。

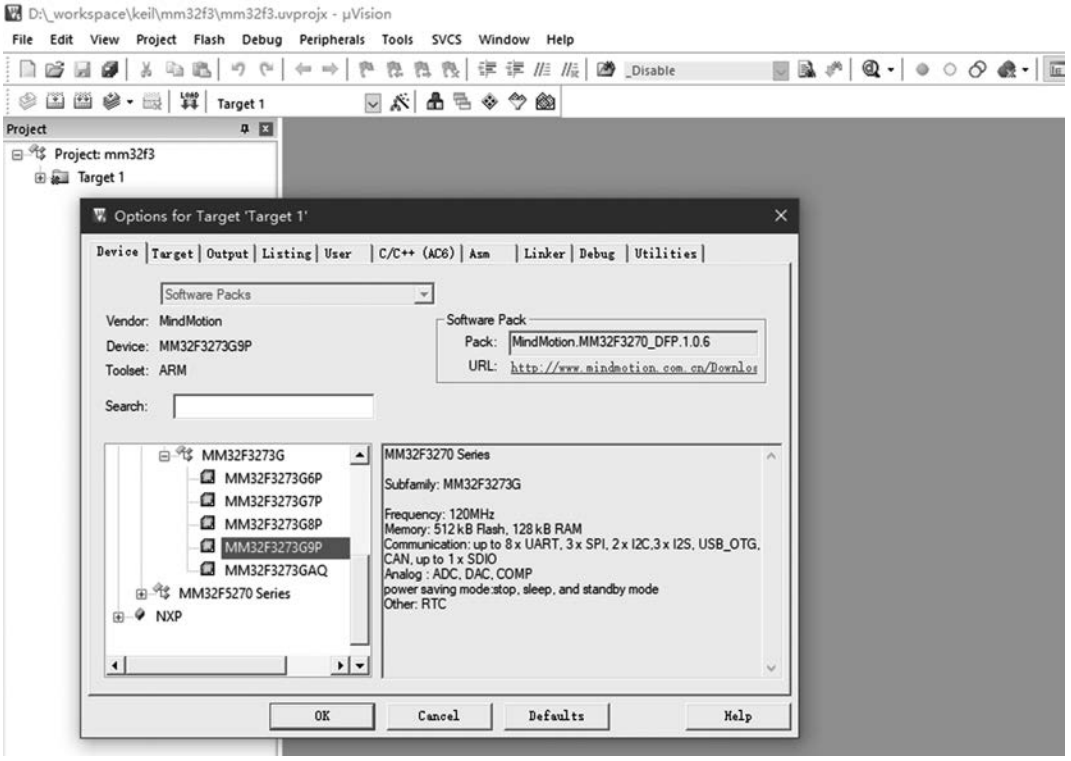

图 3-4　在 Keil IDE 中创建新工程

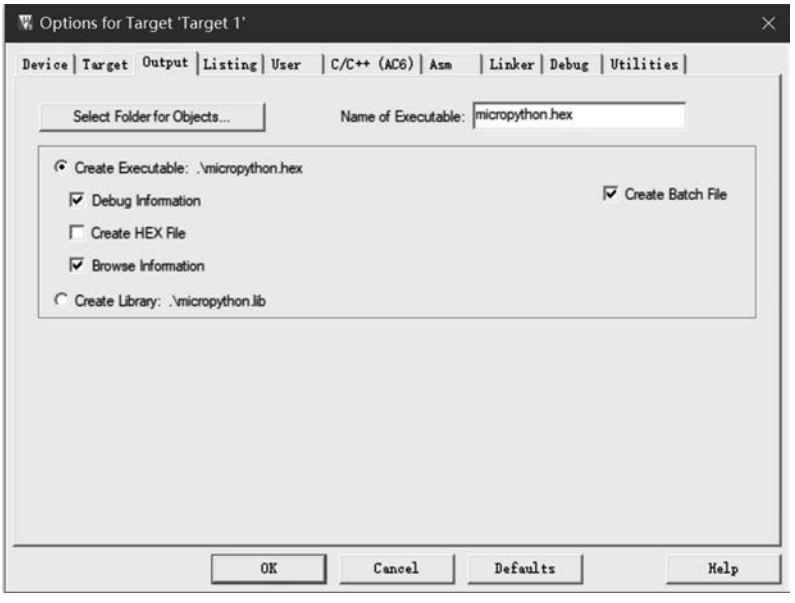

图 3-5　在 Keil 工程中指定下载文件的路径及文件名

(3) 在主窗口的工具栏中,单击 LOAD 按钮,启动下载过程,如图 3-6 所示。在 Build Output 窗口中可以查看到,当前已经擦除、下载并且校验成功。

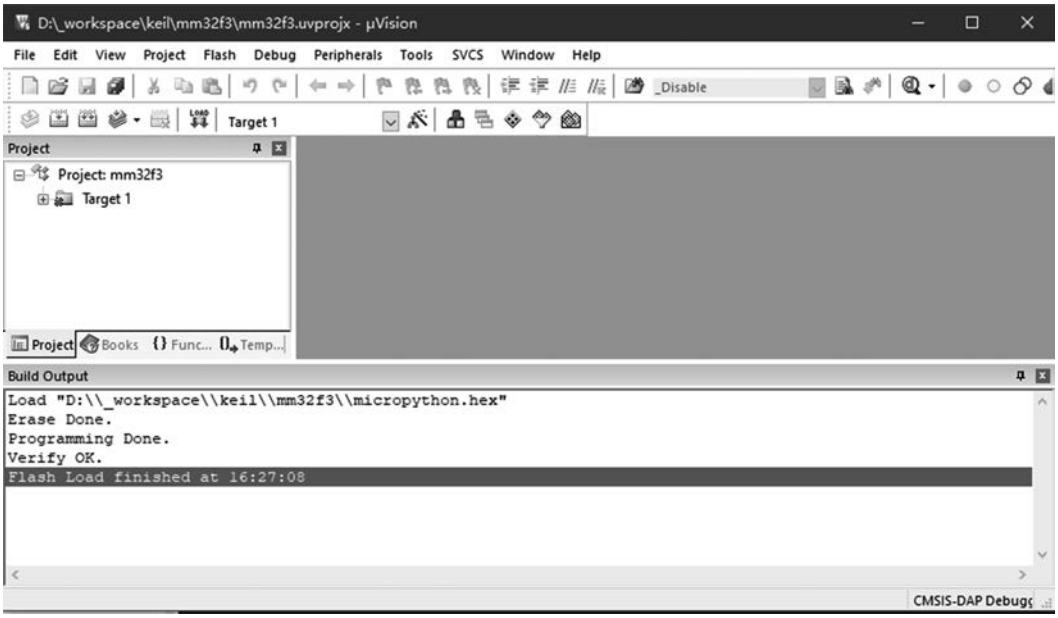

图 3-6　在 Keil 工程中下载可执行文件到微控制器

(4) 也可以使用 Keil 实现命令行式的下载操作。

此时,可将使用 Keil 创建的工程视作 uv4.exe 程序的配置文件,例如,在之前创建的 mm32f3.uvprojx 文件中,编辑 OutputDirectory 和 OutputName 字段,指定将要下载程序的路径和文件名,见代码 3-51。

代码 3-51　mm32f3.uvprojx 源文件

```
<?xml version = "1.0" encoding = "UTF - 8" standalone = "no" ?>
< Project xmlns:xsi = "http://www.w3.org/2001/XMLSchema - instance" xsi:noNamespaceSchemaLocation =
"project_projx.xsd">

  < SchemaVersion > 2.1 </SchemaVersion >

  < Header > ### uVision Project, (C) Keil Software </Header >

  < Targets >
    < Target >
      < TargetName > Target 1 </TargetName >
      < ToolsetNumber > 0x4 </ToolsetNumber >
      < ToolsetName > ARM - ADS </ToolsetName >
      < uAC6 > 1 </uAC6 >
      < TargetOption >
        < TargetCommonOption >
          < Device > MM32F3273G9P </Device >
          < Vendor > MindMotion </Vendor >
          < PackID > MindMotion.MM32F3270_DFP.1.0.6 </PackID >
          < PackURL > http://www.mindmotion.com.cn/Download/MDK_KEIL/</PackURL >
```

```
            < Cpu > IRAM(0x20000000,0x20000) IROM(0x08000000,0x80000) CPUTYPE("Cortex - M3")
CLOCK(12000000) ELITTLE </Cpu >
            < FlashUtilSpec ></FlashUtilSpec >
            < StartupFile ></StartupFile >
            < FlashDriverDll > UL2CM3( - S0 - C0 - P0 - FD20000000 - FC1000 - FN1 - FF0MM32F3270_
512 - FS08000000 - FL080000 - FP0( $ $ Device:MM32F3273G9P $ Flash\MM32F3270_512.FLM))
</FlashDriverDll >
            < DeviceId > 0 </DeviceId >

< RegisterFile > $ $ Device:MM32F3273G9P $ Device\MM32F327x\Include\mm32_device.h </RegisterFile >
            < MemoryEnv ></MemoryEnv >
            < Cmp ></Cmp >
            < Asm ></Asm >
            < Linker ></Linker >
            < OHString ></OHString >
            < InfinionOptionDll ></InfinionOptionDll >
            < SLE66CMisc ></SLE66CMisc >
            < SLE66AMisc ></SLE66AMisc >
            < SLE66LinkerMisc ></SLE66LinkerMisc >
            < SFDFile > $ $ Device:MM32F3273G9P $ SVD\MM32F3270.svd </SFDFile >
            < bCustSvd > 0 </bCustSvd >
            < UseEnv > 0 </UseEnv >
            < BinPath ></BinPath >
            < IncludePath ></IncludePath >
            < LibPath ></LibPath >
            < RegisterFilePath ></RegisterFilePath >
            < DBRegisterFilePath ></DBRegisterFilePath >
            < TargetStatus >
              < Error > 0 </Error >
              < ExitCodeStop > 0 </ExitCodeStop >
              < ButtonStop > 0 </ButtonStop >
              < NotGenerated > 0 </NotGenerated >
              < InvalidFlash > 1 </InvalidFlash >
            </TargetStatus >
            < OutputDirectory >.\</OutputDirectory >
            < OutputName > micropython.hex </OutputName >
            < CreateExecutable > 1 </CreateExecutable >
...
```

然后在 Windows 的命令行界面输入调用 Keil 编译工程的命令，见代码 3-52。

代码 3-52　使用 Keil 命令执行编译

```
uv4.exe - f "d:\_workspace\keil\mm32f3\mm32f3.uvprojx" - j0 - o "d:\_workspace\keil\mm32f3\
download_log.txt"
```

在 Windows 的命令行界面执行 Keil 下载代码的命令，如图 3-7 所示。

使用命令行方式有一点不方便，命令行在后台调用 Keil 执行下载过程没有任何用户交互。命令行触发执行 uv4.exe 程序不是阻塞式的，所以无法通过程序是否返回判定下载过

图 3-7　使用 Keil 命令行下载程序

程已结束。虽然在命令中指定输出 log 到指定文件中，但 Keil 并不是在下载结束后才创建输出文件，而是逐条写入输出文件。如果用户在下载过程中打开这个 log 输出文件，将会看到已经执行的部分操作。因此，不能通过是否创建 log 输出文件判定下载是否成功。必须检查 log 输出文件的内容，待其中包含下载成功并通过验证的记录后，才能最终判定下载情况。

3.3.1.2　使用 Ozone

常用 J-Link 调试器的开发者对 Ozone 都不陌生。Ozone 和 J-Link 都是 SEGGER 公司设计发布的面向调试和下载应用的工具，Ozone 是一套具有图形界面的上位机工具，可以适配 J-Link 调试器，独立下载可执行文件到目标微控制器并进行调试。但实际上，Ozone 除了适配自家发售的 J-Link 调试器外，还提供了对开源 CMSIS-DAP（DAP-Link）的支持，即使用 Ozone 通过 DAP-Link 连接到目标微控制器，也能够实现独立下载并执行和调试文件的功能。不过，Ozone 支持 DAP-Link 毕竟只是额外的福利，所以几乎每个步骤都会在弹窗中提示："这只是个试用功能，未经过充分测试。"

使用 Ozone 适配 DAP-Link 的操作同使用 J-Link 的情况相同，具体步骤如下：

（1）启动 Ozone 软件，选择目标微控制器设备。

刚启动 Ozone 软件时，Ozone 会自动检测到当前电脑上已经接入了 DAP-Link 调试器，然后提示警告"必须接受如下条款：1. 当前软件仅适用非商业用途或评估；2. SEGGER 官方不会提供技术支持"。单击 Accept 按钮，如图 3-8 所示。

（2）选择目标微控制器设备。

当确认目标微控制器设备后，警告提示对话框会再次弹出。仍然是单击 Accept 按钮，如图 3-9 所示。

（3）在连接配置对话框中，可以看到已经识别出来的 DAP-Link，如图 3-10 所示。

Ozone 连接调试器的速度，在默认情况下被配置成 4MHz。在作者的 DAP-Link 方案中，使用的是低速 USB 接口，为了更稳妥，将速度改为 1MHz。

（4）选择将要下载的可执行文件，如图 3-11 所示。

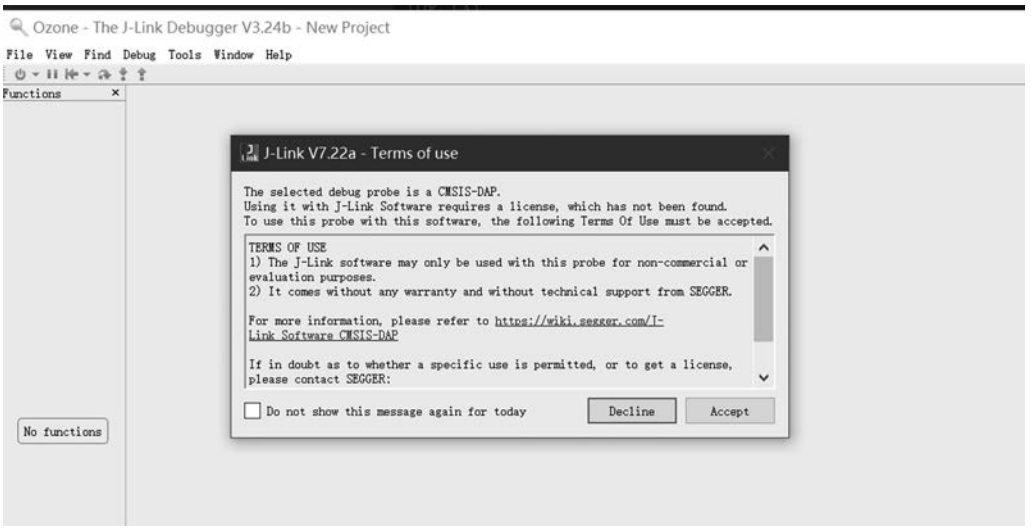

图 3-8　启动 Ozone

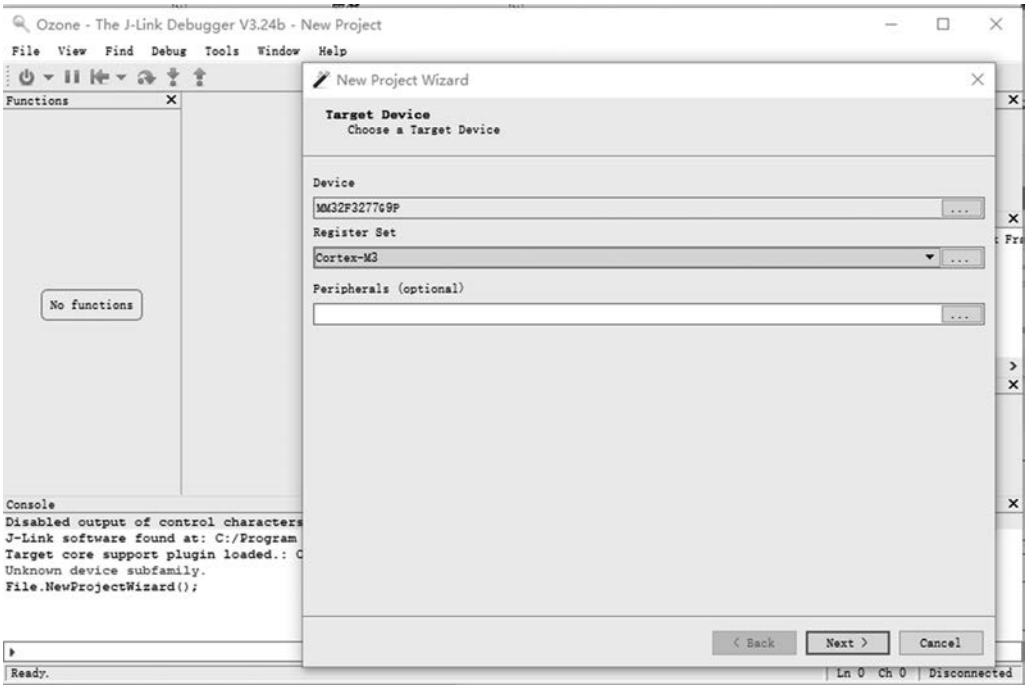

图 3-9　在 Ozone 中选择目标设备

(5) 开始下载。

弹出提示对话框,单击 Accept 按钮,如图 3-12 所示。

再次弹出提示对话框,单击 Yes 按钮。

(6) 下载成功,如图 3-13 所示。

如果不想看到频繁弹出的警告对话框,则可选中“不要重复弹出”复选框,有一定的改善效果。

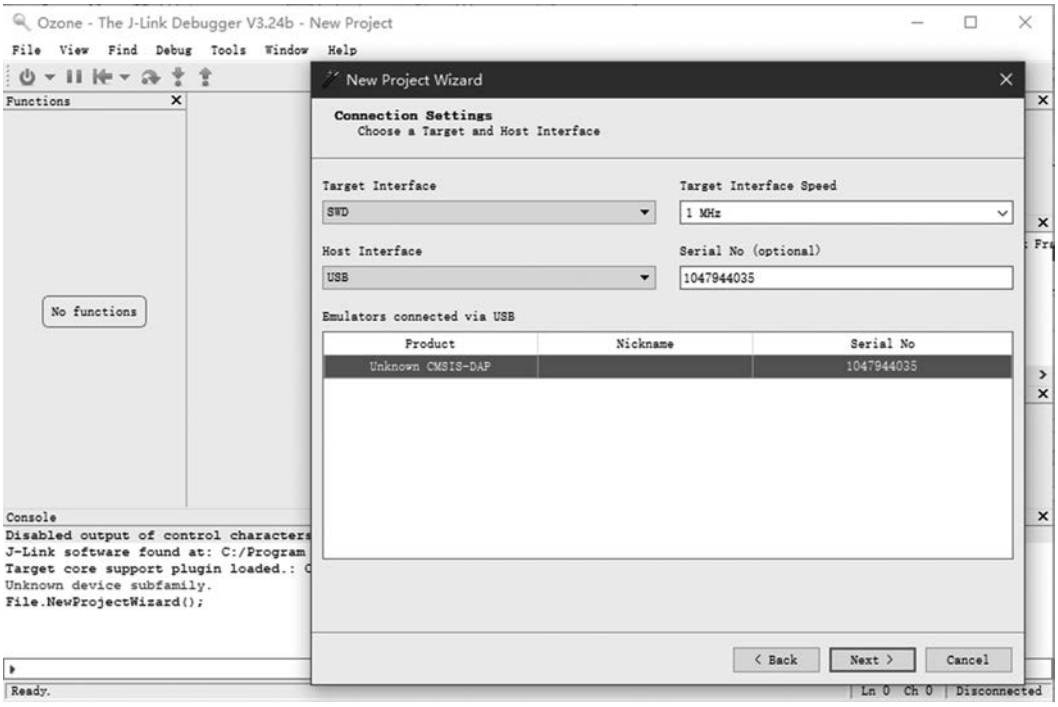

图 3-10　Ozone 显示识别出的 DAP-Link 调试器

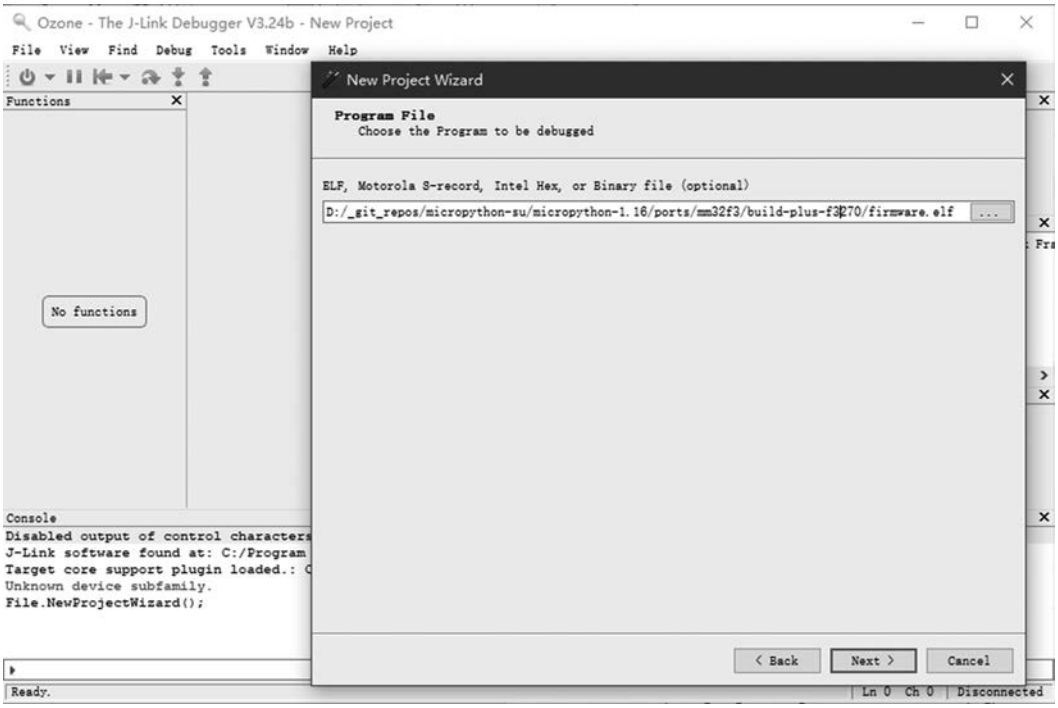

图 3-11　选择将要下载的可执行文件

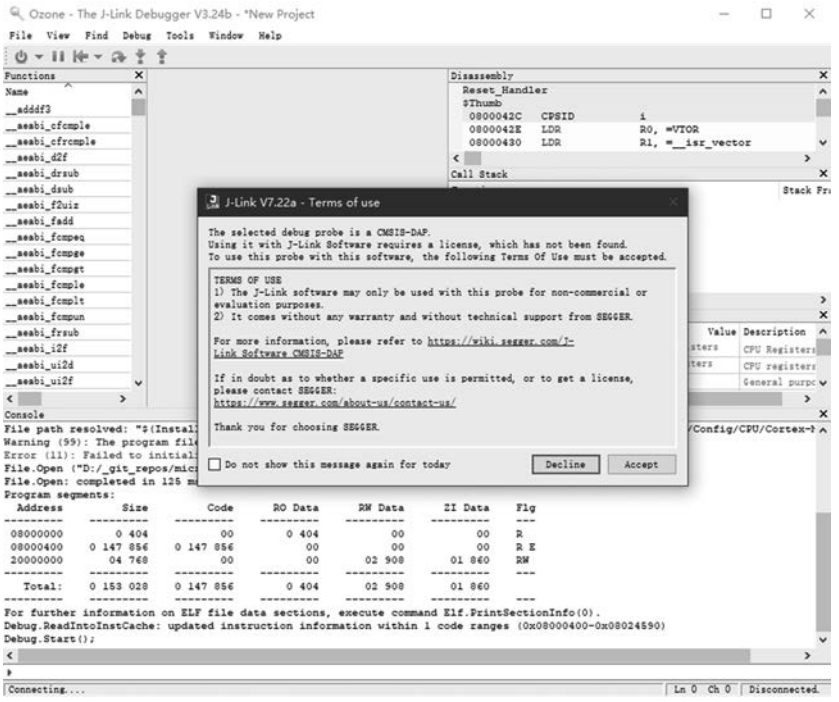

图 3-12　Ozone 准备下载固件

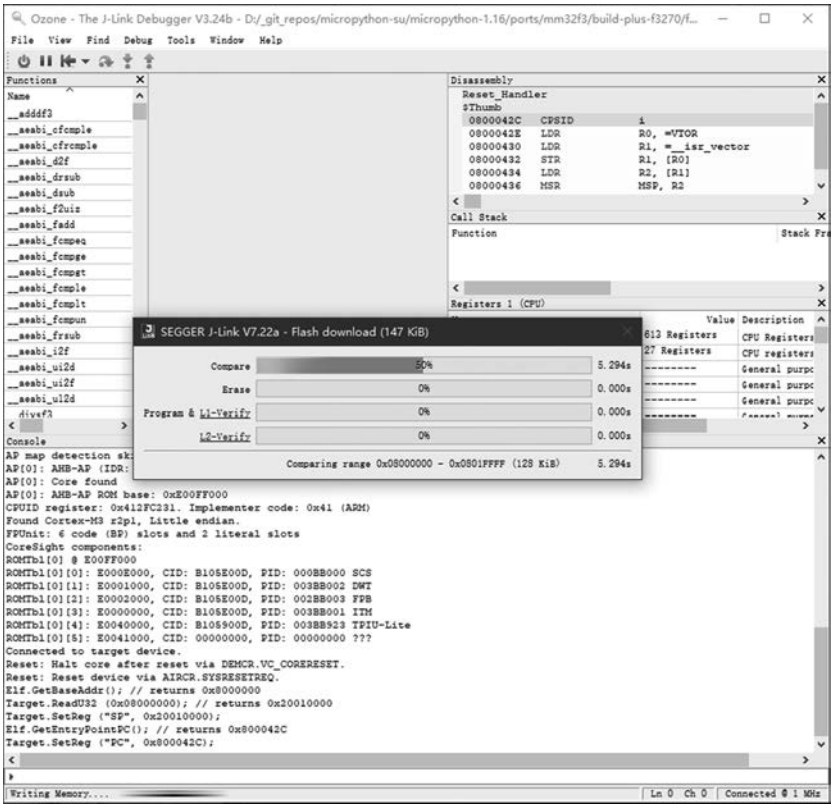

图 3-13　使用 Ozone 下载成功

3.3.2　验证及演示程序

下面介绍在 MM32F3 微控制器上运行 Python 内核。

将 MicroPython 内核编译下载到 MM32F3270 微控制器后，可以通过以下实验初步验证硬件平台和 MicroPython 内核软件工作正常。

注意，在本书用例的完整移植项目中已经包含了 Thonny 集成开发环境的支持，但读者若是从零开始开发，那么在 minimal 工程中，尚未支持 Thonny 集成开发环境（需要适配文件系统、os 模块和 time 模块等），因此仅能通过 REPL 同 MicroPython 内核建立通信。

将 MicroPython 硬件平台（比如 PLUS-F3270、MM32F3270 最小系统实验板）通过 USB 接入计算机，可以识别 USB 模拟出的串口。使用 Tera Team 等串口通信终端工具软件可以连上识别的串口，建立 REPL 通信。

MicroPython 内核通过 REPL 与用户进行交互。将 MicroPython 硬件上电后，可以观察到通信终端的信息窗口显示启动 MicroPython 内核成功的提示信息，见代码 3-53。

代码 3-53　启动 MicroPython 内核成功的提示信息

```
MicroPython v1.16 on 2022-06-29; PLUS-F3270 with MM32F3273G9P
>>>
```

试着执行一个简单的加法运算和 for 循环语句，在终端窗口中输入脚本并执行，如图 3-14 所示。

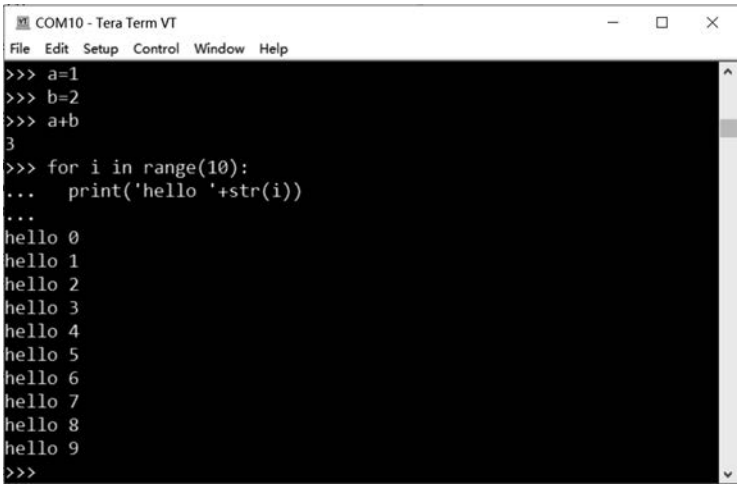

图 3-14　在 REPL 中运行 Python 脚本

由此验证，MicroPython 可以在 MM32F3 微控制器上运行。

3.4　本章小结

如前所述，准备好开发 MicroPython 的软硬件开发环境之后，本章在 MM32F3 微控制器平台上简要分析并移植了 MicroPython 自带的 Minimal 最小工程。移植过程涉及向

MicroPython 源码目录中添加 MM32F3 的 SDK 启动源文件和驱动程序、为新的微控制器平台创建移植项目目录结构、改写部分与移植相关的源文件、调整 main.c 中的执行流程以及更新 Makefile 文件。经过一系列的工作，读者可以在预先搭建好的编译环境中执行编译，并创建 MM32F3 微控制器平台的 MicroPython 固件文件。本章还介绍了实现单独下载 armgcc 工具链创建的可执行文件到 MM32F3 微控制器中的多种方式。

终于可以在 MM32F3 微控制器上运行 MicroPython 啦！试着通过 UART 串口连上 MicroPython 的 REPL，输入一些 Python 语句，开始体验使用 Python 进行单片机开发的乐趣吧。