第 3 章

CHAPTER 3

STM32 的开发工具介绍及安装

为了开发 STM32，需要准备代码开发环境和连接硬件用的调试下载工具。下面分别介绍这两类工具。

为了将编写的代码编译成可下载到芯片中的文件，需要利用开发环境对编写的代码进行编译。常见的开发环境有 MDK、EWARM、STM32CubeIDE、GNU 编译器套件（GNU Compiler Collection，GCC）等。

MDK 全称为 Microcontroller Development Kit，即微控制器开发套件。它源自 Keil 公司（后被 ARM 公司收购），现也称为 MDK-ARM。MDK-ARM 软件为基于 Cortex-M、Cortex-R4、ARM7、ARM9 等内核的处理器设备提供了一个完整的开发环境。

EWARM 全称为 Embedded Workbench for ARM，是 IARSystems 公司为 ARM 微处理器开发的一个集成开发环境。其包含项目管理器、编辑器、编译连接工具和支持 RTOS 的调试工具。在该环境下可以使用 C/C++和汇编语言方便地开发嵌入式应用程序。

STM32CubeIDE 由意法半导体公司开发，是面向 STM32 的一体化集成开发环境。该 IDE 基于 Eclipse 或 GNU C/C++工具链等开源解决方案，包括编译报告功能和高级调试功能。它还额外集成了生态系统中其他工具才有的功能，如来自 STM32CubeMX 的硬件和软件初始化及代码生成功能。其通过多种增强功能（如数据变量实时观察和特殊寄存器视图）帮助快速调试应用程序，代码编辑、项目构建、板级烧录和调试均集成在一起，可实现无缝、快速的开发周期。

GNU 编译器套件是由 GNU 开发的编程语言编译器。前面介绍的几种开发环境属于集成开发环境，它们把代码编辑界面（文本编辑器）、编译器和调试器整合到一个软件中，使得开发起来较为方便。而 GCC 只负责其中的一个环节，即编译器功能。因此，其不包含文本编辑界面，并且使用起来需要配合 Makefile 编译规则说明文件和命令行的各种指令。

为了将开发环境编译生成的文件下载到芯片中并进行调试，还需要调试下载工具将计算机和芯片进行连接。JLINK 和 ST-LINK 是两种常用的用于调试和编程 ARM 微控制器的仿真器，它们都可以通过 USB 接口与计算机连接，实现对目标芯片的内存、寄存器、外设等的访问和控制。

JLINK 是德国 SEGGER 公司推出的仿真器，如图 3.1 所示。它实际上是一个小型 USB 到 JTAG/SWD 的转换盒，其与计算机通过 USB 接口连接，采用的是 JTAG 协议或 SWD 协议。JLINK 还支持广泛的 CPU 和体系结构，从 8051 到大众市场的 Cortex-M，再到 Cortex-A（32 位和 64 位）等高端内核。JLINK 支持直接连接 SPI 闪存，不需要

图 3.1　JLINK

在JLINK和SPI闪存之间使用CPU(通过SPI协议直接通信)。JLINK得到了多个主流集成开发环境的进一步支持,包括SEGGER Embedded Studio、Visual Studio Code、Keil MDK、IAR EWARM等。

ST-LINK是意法半导体公司为开发STM8/STM32系列MCU而设计的集在线仿真与下载功能于一体的开发工具,支持SWIM、JTAG、SWD共3种模式。最新的ST-LINK型号为V3,提供了一个虚拟COM端口,允许主机PC通过一个UART与目标微控制器通信,以及桥接接口(SPI、I^2C、CAN、GPIO),允许通过引导加载程序对目标进行编程。其支持的开发环境有MDK-ARM、IAR EWARM、基于GCC的集成开发环境。

ARM Mbed DAPLink是一个开源软件项目,用于在ARM Cortex CPU上运行的应用软件上进行编程和调试。DAP全称为Debug Access Port,即调试访问端口。它是一种开源的调试与编程接口协议,可以被下载使用。该项目正在由ARM、其合作伙伴、众多硬件供应商以及全球开源社区进行持续开发。DAPLink已经取代了mbed CMSIS-DAP接口固件项目。

本书主要介绍MDK+DAPLink的开发方式,同时使用STM32CubeMX进行初始的工程项目配置和生成。


视频讲解

3.1 生成工程模板——STM32CubeMX

STM32CubeMX是一种图形化的配置工具,通过分步过程可以非常轻松地配置STM32微控制器和微处理器,以及为ARM Cortex-M内核或面向ARM Cortex-A内核的特定Linux设备树生成相应的初始化C代码。在STM32CubeMX诞生以前,需要手工一步一步地在MDK中引入工程文件,并且手工添加外设配置代码。在使用STM32CubeMX以后,我们只需要在软件中选择STM32的相应功能,即可让其自动生成属于MDK的配置好外设的工程模板。

3.1.1 STM32CubeMX的安装

下面介绍该软件的安装过程。

(1) 由于STM32CubeMX的使用需要Java开发环境,因此需要先去Java的官方网站上获取Java开发环境的安装包。打开页面后选择下载Java,下载的文件名类似于jre-8u401-windows-x64.exe。

(2) 打开下载的Java安装包进行安装,如图3.2所示。

图3.2 Java安装界面

（3）从意法半导体公司官方网站上获取 STM32CubeMX 安装包（“首页”→“工具与软件”→“开发工具”→“STM32 软件开发套件”→“STM32 配置程序和代码生成器”→STM32CubeMX），如图 3.3 所示。在页面的下方选择属于不同系统的安装包进行下载（下载前需注册登录）。

图 3.3　STM32CubeMX 下载页面

（4）运行 STM32CubeMX 的安装包进行安装。注意：安装目录不可出现中文。

（5）安装完成后桌面会出现 STM32CubeMX 的图标。

3.1.2　固件包的安装

不同的芯片需要使用不同的固件来进行配置，因此在配置 STM32 之前还需要在 STM32CubeMX 中下载相应的固件包，如图 3.4 所示。在该软件中，可以通过离线的方式进行固件包的安装，即按照 2.3.1 节介绍的方法一先从官方网站上下载对应的固件包，然后选择软件菜单栏的 Help→Manage embedded software packages 命令，在打开的窗口中单击 From Local 按钮。注意，需要先安装基础包（如 1.2.0），再安装扩展包（如 1.2.1）。这里推荐采用在线的方法自动进行安装，即 2.3.1 节介绍的方法二。

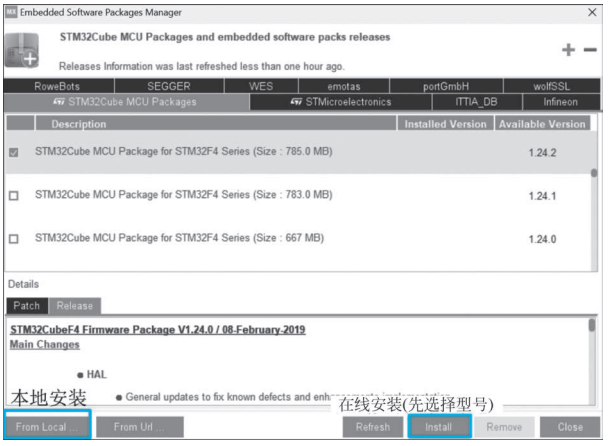

图 3.4　STM32CubeMX 中的固件包安装界面

由于后续实验使用 STM32U575，这里选择 STM32U5 的最新固件进行安装。安装完成后，执行 Help→Manage embedded software packages 命令，在打开的窗口中，对应的固件包版本前面的方框会变为绿色，如图 3.5 所示。

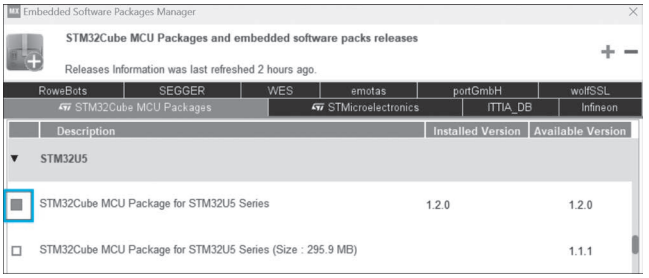

图 3.5　固件包安装完成

3.1.3　配置并生成代码模板

(1) 打开 STM32CubeMX,选择菜单栏的 File→New Project 命令。

(2) 在弹出窗口的左上角输入想要配置的芯片的型号,在 MCUs/MPUs List 中双击对应的型号即可打开配置窗口,如图 3.6 所示。

由于 STM32U575 支持 TrustZone 功能,因此会弹出如图 3.7 所示的对话框,单击 OK 按钮即可,默认不选择 TrustZone 功能。其他一些型号不会弹出此对话框。

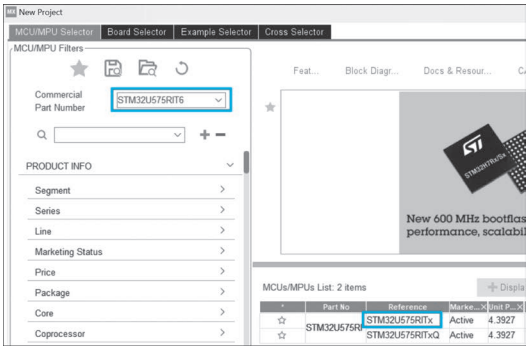

图 3.6　芯片型号选择界面

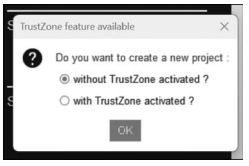

图 3.7　选择是否启用 TrustZone

(3) 在配置窗口的 Pinout & Configuration 标签页中,左侧可以设置各种外设及功能的参数,右侧可以配置每个引脚的工作模式,右下角的放大镜处用来搜索引脚的编号,以便定位引脚,如图 3.8 所示。

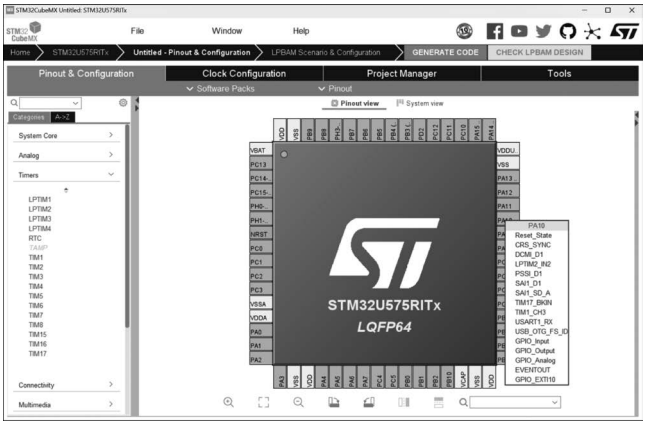

图 3.8　Pinout & Configuration 标签页

在 Clock Configuration 标签页中，可以配置各个总线上的时钟来源和频率。

在 Project Manager 标签页中，可以配置工程的相关设置，如目录、代码支持的 IDE、固件包版本号、生成的文件列表等。

Tools 标签页中是其他的一些工具，如 PCC 代表 Power Consumption Calculator，可用于计算能耗；CAD 允许用户快速访问和下载一个或多个设计工具链的原理图符号、PCB 示意图和 3D CAD 模型。

(4) 设置好之后，单击 GENERATE CODE 按钮即可在对应的目录生成对应 IDE 使用的工程模板。详细的代码生成步骤及编译、调试、下载方法将会在实验中进行说明。

有关 STM32CubeMX 的详细使用方法，可单击菜单栏的 Help 进行了解。

3.2　编辑编译工程——MDK-ARM、STM32CubeIDE


视频讲解

3.2.1　MDK-ARM

下面介绍该软件的安装过程。

(1) 由于最新的 MDK 不支持 Compiler Version 5 编译器，而利用 STM32CubeMX 生成的工程需要使用此编译器，因此需要下载 5.37 版本以前的 MDK。打开 Keil 官方网站的 ARM Product Updates 页面，在页面中找到 5.36 版本的下载链接并单击打开，如图 3.9 所示。

(2) 在打开的页面中，需要填写相关信息才能下载。

(3) 双击安装程序安装即可。注意，要安装在全英文目录下，同时设置好程序内核的安装位置和固件包的安装位置，如图 3.10 所示。

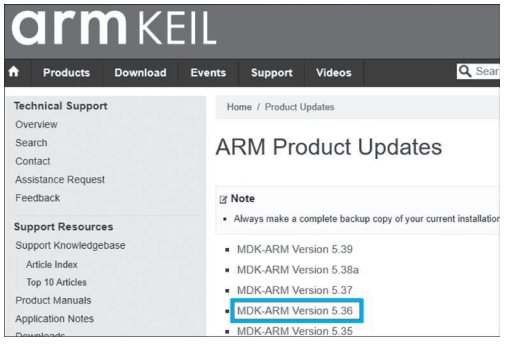

图 3.9　ARM Product Updates 页面

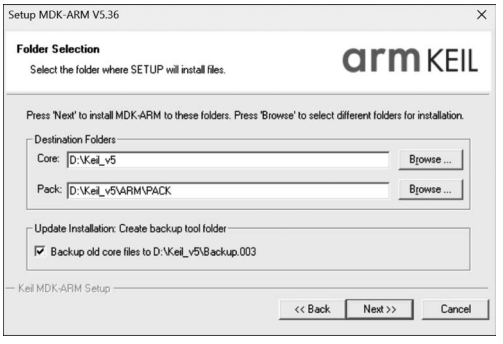

图 3.10　安装程序目录配置界面

(4) 安装完成后，桌面上会出现软件图标，如图 3.11 所示。

(5) 打开软件后，需要先选择菜单栏的 File→License Management 命令进行认证后才能使用该软件的全部功能，否则编译文件时会受到大小等限制。请咨询官方获取注册码。

图 3.11　MDK 图标

(6) 软件安装完成后，还需要安装针对特定芯片的固件包。该固件包和 STM32CubeMX 中安装的固件包不同，它是应用于 MDK 的，因此需要独立下载安装。打开 ARM Keil 官方网站，在页面上方找到 CMSIS Packs 并打开，如图 3.12 所示。

(7) 由于本书主要以 STM32U5 为例进行讲解，因此这里按照图 3.13 进行搜索，然后单击结果中的 STM32U5xx_DFP Keil。

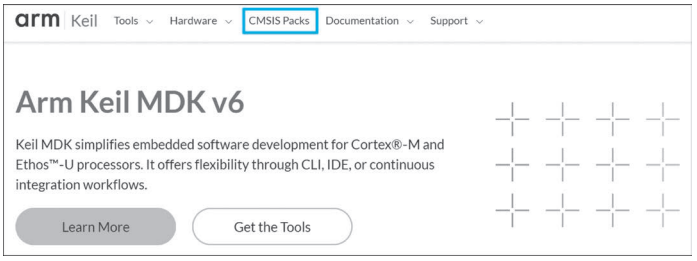

图 3.12　ARM Keil 官方网站

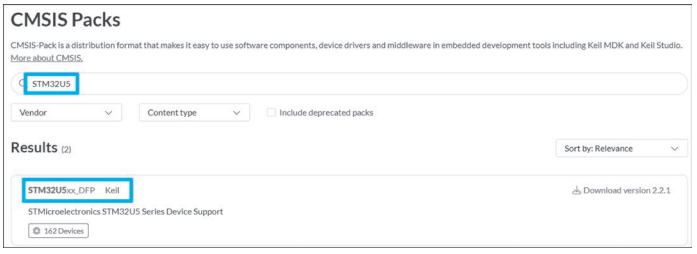

图 3.13　MDK 固件包搜索页面

（8）在 Version History 中选择一个较老的版本，以便适应老版本的 MDK，如图 3.14 所示。这里选择 2.1.0 版本，如图 3.15 所示。

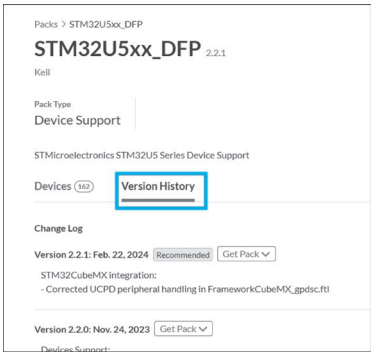

图 3.14　固件包历史版本页面

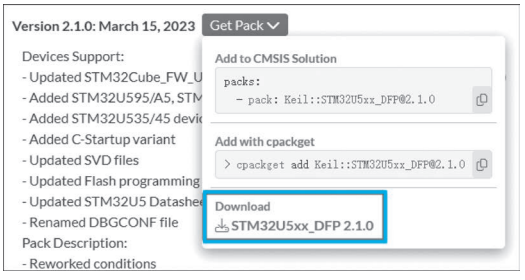

图 3.15　STM32U5xx_DFP 2.1.0 固件包下载页面

（9）打开 Keil 软件的安装目录，在 Keil_v5\UV4 文件夹下找到 PackInstaller.exe，右击该图标，选择以管理员身份运行。

（10）由于默认启动后会检查更新，影响后续操作，因此需要先取消选中“启动后自动更新”选项，即取消选中 Packs→Check For Updates on Launch，如图 3.16 所示。取消选中后再以管理员身份运行 PackInstaller.exe。

（11）在 Pack Installer 中，选择 File→Import 命令，如图 3.17 所示。然后在打开的窗口中选择下载的安装包 Keil.STM32U5xx_DFP.2.1.0.pack 进行安装。

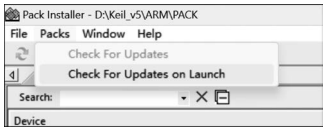

图 3.16　启动后检查更新选项

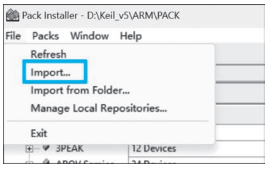

图 3.17　Pack Installer 的 File 菜单

注意，MDK 版本要和 Packs 安装包版本对应。有些老版本的 MDK 可能会在打开新版本的固件包时产生错误，因此可以在版本历史中下载旧版本的固件包，如 1.2.0 版本。

（12）安装成功后，选择相应的芯片，即可在右侧显示固件包已安装，如图 3.18 所示。

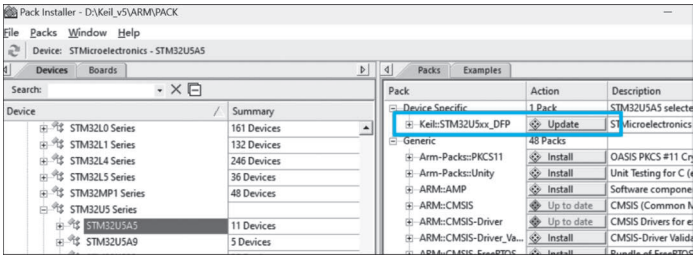

图 3.18　MDK 的固件包安装状态

3.2.2　STM32CubeIDE

STM32CubeIDE 是意法半导体公司官方提供的免费软件开发工具，也是 STM32Cube 生态系统的一员大将。它基于 Eclipse/CDT 框架、GCC 编译工具链和 GDB 调试工具，支持添加第三方功能插件。同时，STM32CubeIDE 还集成了部分 STM32CubeMX 和 STM32CubeProgrammer 的功能，是一个多合一的 STM32 开发工具，如图 3.19 所示。

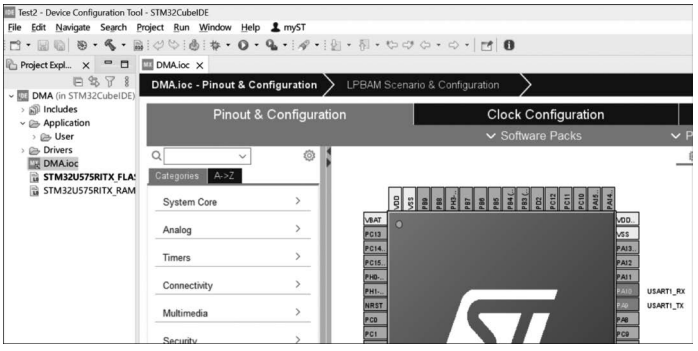

图 3.19　STM32CubeIDE 软件界面

用户只需要 STM32CubeIDE 这一个工具，就可以完成从芯片选型、项目配置、代码生成到代码编辑、编译、调试和烧录的所有工作。

在开发过程中，用户也可以非常方便地切换到内嵌的 STM32CubeMX 初始化窗口，添加或者修改之前的外设和中间件配置。不需要在多个工具之间进行切换。

STM32CubeIDE 提供的编译和堆栈分析工具为用户提供了关于项目状态和内存使用的有用信息，还提供了很多高级的调试功能帮助用户进行高效调试。

与 STM32CubeMX、STM32CubeProgrammer 一样，STM32CubeIDE 也是一个多平台的 STM32 开发工具，用户可以在 Windows、Linux 和 macOS 操作系统上通过 STM32CubeIDE 进行软件开发。

下面介绍该软件的安装过程。

（1）按照 3.1.1 节的步骤安装 Java 环境。

（2）打开意法半导体公司官方网站，依次进入“工具与软件”→“开发工具硬件”→“软件

开发工具”→“STM32 软件开发套件”→“STM32 软件开发套件”→STM32CubeIDE，如图 3.20 所示。在页面下方选择对应系统的版本下载即可（下载前需注册登录）。

图 3.20　STM32CubeIDE 下载页面

（3）注意设置安装目录为全英文目录，如图 3.21 所示。

（4）选择安装两个仿真器驱动，如图 3.22 所示。

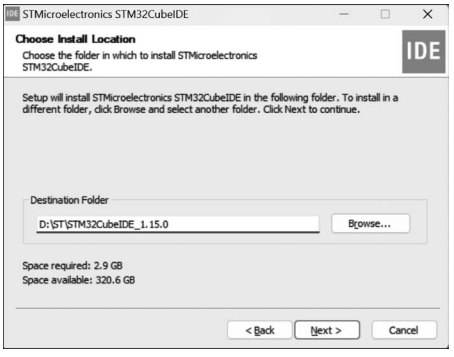

图 3.21　STM32CubeIDE 安装目录设置

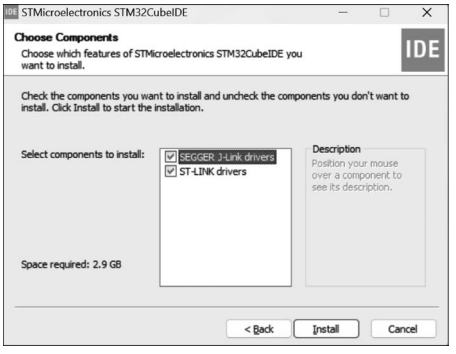

图 3.22　STM32CubeIDE 驱动设置

（5）安装完成后在桌面上会显示对应的图标。

打开工程后，窗口各部分功能如图 3.23 所示。

图 3.23　STM32CubeIDE 界面

工具栏如图 3.24 所示。

图 3.24　STM32CubeIDE 的工具栏

使用图标 创建新 C 源代码文件、头文件或新目标，例如，工程、库或存储集（执行主菜单中的 File→New 命令）。

使用图标 编译工程。

使用图标 启动调试，或者单击箭头对调试配置进行设置（此功能可通过主菜单中的 Run 选项启动）。

手电筒图标 用于启动各种搜索工具，利用箭头 可浏览最近访问的工程区域（对应菜单栏中的 Search 和 Navigate）。

3.3　调试下载工具

视频讲解

3.3.1　ST-LINK

ST-LINK 硬件主要由两部分组成：ST-LINK 主控板和连接线。ST-LINK 主控板上有一个 USB 接口、一个 20 针 JTAG/SWD 连接口和一个 LED 指示灯。USB 接口用于连接计算机，JTAG/SWD 连接口用于连接芯片进行调试和编程。各版本 ST-LINK 如图 3.25 所示。

ST-LINK V1　　ST-LINK V2　　ST-LINK V3MINI

图 3.25　各版本 ST-LINK

ST-LINK V1 是较老的版本，官方网站上显示已经停产。

ST-LINK V2 是目前比较常见的版本，相比于 V1 有更高的数据传输速率。

ST-LINK V3 是针对 STM8 和 STM32 的新一代模块化在线调试兼编程功能的工具。STLINK-V3 包含 3 个版本：STLINK-V3SET、STLINK-V3MINI、STLINK-V3MODS。V3 相较于 V2 有更高的数据传输速率，同时具备更高灵活性和扩展性，满足定制化需求。

为了在计算机上正常使用 ST-LINK，需要为其安装驱动。驱动程序下载页面如图 3.26 所示。

图 3.26　ST-LINK 驱动程序下载页面

下载并解压后，如果系统是 64 位的，则右击 dpinst_amd64.exe 后选择以管理员身份运行即可安装。如果系统是 32 位的，则右击 dpinst_x86.exe 后选择以管理员身份运行即可安装。

驱动安装完成后，在计算机上插入 ST-LINK，设备管理器中会显示相应的设备，如图 3.27 所示。

图 3.27 设备管理器中的 ST-LINK 设备

3.3.2 DAPLink

图 3.28 是基于 ARM Mbed DAPLink 的开源项目开发的一款 DAPLink 调试下载器，可以看到其上部为 USB 接口，下部为 10 针的调试下载引脚。

DAPLink 是免驱的，插上设备即可使用，如图 3.29 所示。

图 3.28 一款 DAPLink

图 3.29 设备管理器中的 DAPLink 设备

本书的实验使用 DAPLink 进行程序的下载和调试，也可以使用 ST-LINK，只需调整 MDK 中的工程选项即可，如图 3.30 所示。

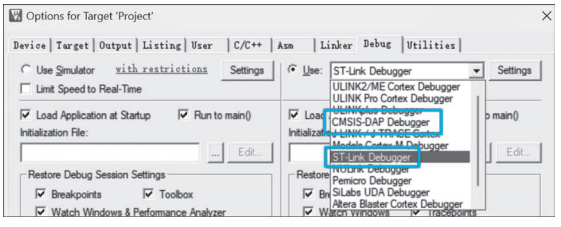

图 3.30 MDK 中对调试下载器的选择

3.4 串行通信工具

为了在计算机和单片机之间进行串行通信，既需要在单片机侧连接串行通信芯片，也需要在计算机上安装对应芯片的驱动程序。

CH340 是一款 USB 转串口的芯片。通过将 CH340 部署到开发板上，可以实现计算机与开发板的串行通信。因此，需要在计算机上安装 CH340 驱动。

打开 WCH 官方产品页面，在页面下方找到驱动程序，选择对应的系统下载即可，如图 3.31 所示。这里下载 CH341SER.EXE，因为它与 CH340 驱动兼容。

运行下载的安装程序后，单击“安装”按钮。

安装成功后，当利用 USB 线连接含有 CH340 的开发板时，可以看到设备管理器里面能够识别到 CH340，如图 3.32 所示。

图 3.31　CH340 官方驱动下载页面

图 3.32　设备管理器中识别到的 CH340

在开发过程中，可以在计算机上使用串口调试助手软件来与开发板进行串行通信。

3.5　STM32 硬件开发平台

为了对 STM32 进行编程并调用各种外设接口、传感器，需要使用特定的硬件开发平台进行学习。本书使用由华清远见研发的 FS-STM32U5 开发板。FS-STM32U5 开发板由底板、核心板、显示屏、资源扩展板组成，如图 3.33 所示。该开发板可更换 STM32F407、STM32F103 等型号的核心板，以便学习其他型号的 STM32。

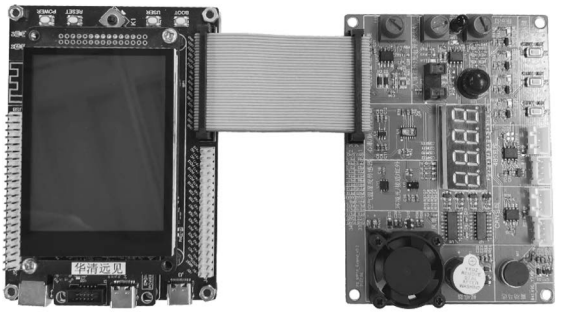

图 3.33　FS-STM32U5 开发板

3.6　实验：用 STM32CubeMX 和 MDK 创建工程项目并调试

视频讲解

本实验将使用在前面章节中安装好的开发环境进行工程的创建和调试，从而熟悉利用 STM32CubeMX 和 MDK 开发 STM32 的流程，并熟悉相关软件的使用方法。本实验的实验效果是：在主函数中创建变量后，在主循环中为此变量加 1，通过在 Debug 模式中插入断点并查看执行效果。

3.6.1　配置 STM32CubeMX 工程

(1) 打开 STM32CubeMX，选择菜单栏的 File→New Project 命令新建一个工程。在弹出的页面中搜索对应的型号。本实验使用 STM32U575RIT6(如果使用其他型号，则搜索对应型号，然后继续后续步骤即可)。在 MCUs/MPUs List 中找到对应型号后双击打开，如图 3.34 所示。

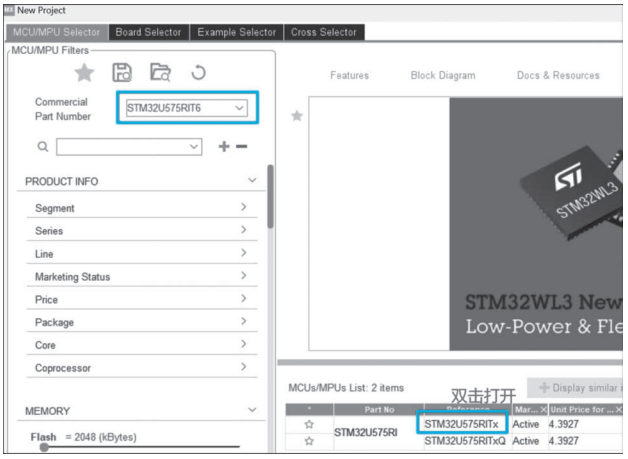

图 3.34　选择 STM32 型号

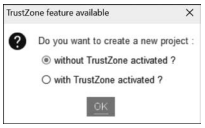

图 3.35　TrustZone 使能选择

（2）由于本芯片支持 TrustZone 功能，所以会弹出如图 3.35 所示页面，选择默认的不使用 TrustZone 选项，单击 OK 按钮。

（3）检查 SYS 界面下是否有 Debug 设置，若有则选择 Serial Wire 模式，若没有该设置则可以不做设置，如图 3.36 所示。

这对应于 ST-LINK 在 MDK 中的调试模式，如图 3.37 所示。

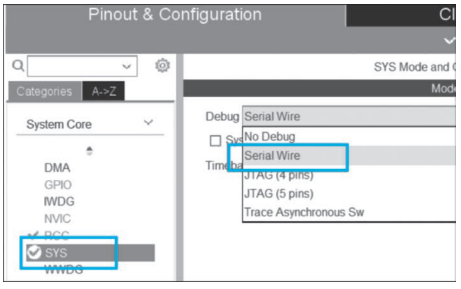

图 3.36　Debug 模式选择

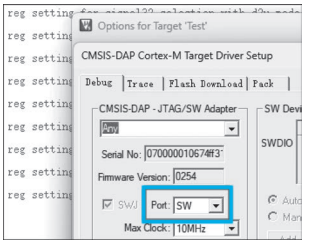

图 3.37　MDK 中的调试接口设置

（4）由于不使用外设，所以直接进入 Project Manager 页面设置工程即可。在 Project 栏目中，需要设置工程名称、工程目录以及使用的 Toolchain/IDE。注意，目录中必须为全英文。本实验使用 MDK-ARM 进行编译，选择默认的版本号即可，如图 3.38 所示。

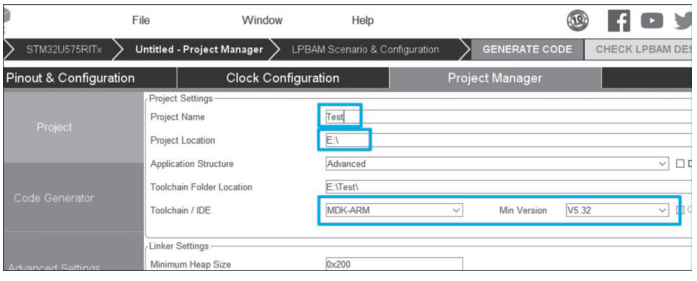

图 3.38　工程设置页面

（5）在 Code Generator 栏目中，选中 Copy only the necessary library files 单选按钮，即只复制必要的文件，以便节省工程占用空间，提高生成速度，同时选中 Generate peripheral initialization as a pair of '. c/. h' files per peripheral 复选框，为外设分别生成. c 和. h 文件，如图 3. 39 所示。

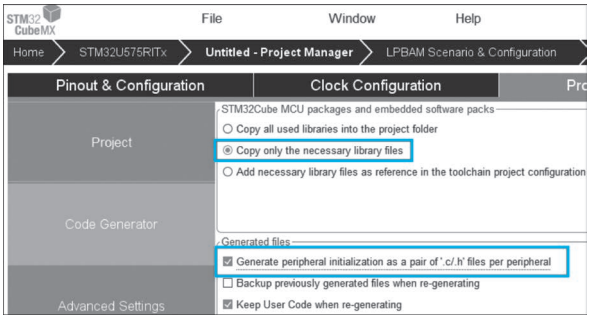

图 3. 39　代码生成设置

（6）单击 GENERATE CODE 按钮。由于 STM32U5 系列支持 ICACHE（Instruction Cache）指令缓存技术，所以会弹出警告对话框提示没有使用 ICACHE。如果不使用高速外设，单击 Yes 按钮即可，如图 3. 40 所示。

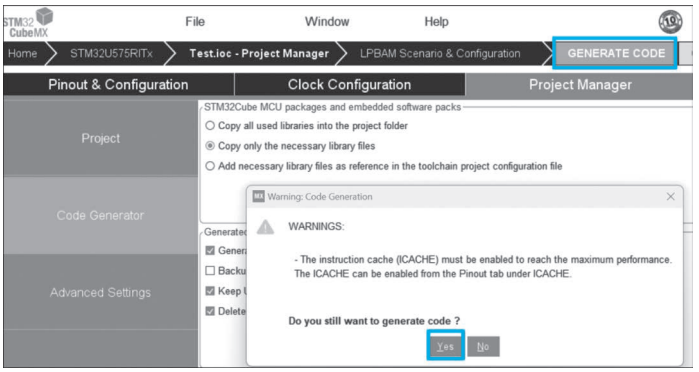

图 3. 40　ICACHE 使能提示

（7）在弹出的对话框中单击 Open Project 按钮即可在 MDK 中打开该工程，如图 3. 41 所示。也可以打开工程所在的文件夹，再双击打开 *. uvprojx 文件。完成此步骤的前提是 MDK-ARM 安装成功并安装好了 STM32U5 的固件包。

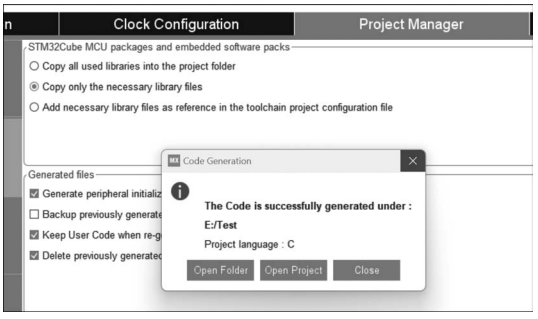

图 3. 41　打开工程选择

3.6.2 使用 MDK 打开并编辑工程

(1) 在打开的工程中,左侧为引入工程的文件,如图 3.42 所示。startup_stm32 **** xx.s 文件为单片机上电后执行的第一段程序,大致分为几个步骤:初始化堆栈指针(SP)、初始化程序计数器(PC)、初始化中断服务程序(ISR)向量表、跳转到__main。system_stm32u5xx.c 文件主要用于系统初始化、系统时钟配置。

(2) 如图 3.43 所示,单击 Options for Target 按钮,打开工程配置界面。

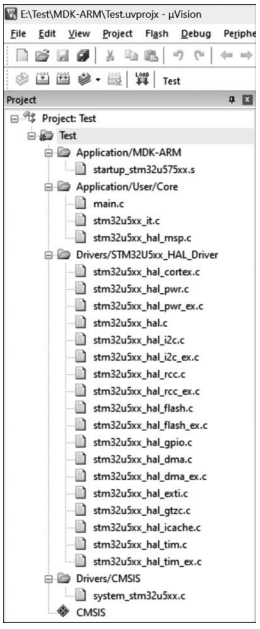

图 3.42 工程文件

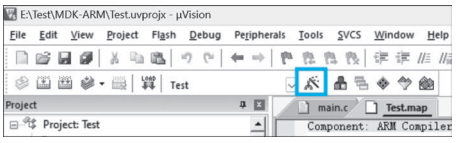

图 3.43 Options for Target 按钮所在位置

Device 标签页可显示选择的芯片型号和固件包版本。

Target 标签页用于设置代码的 ROM 与 RAM 的起始地址、编译器版本等,如图 3.44 所示。MicroLib 是标准 C 库的裁剪版本,一般用于存储空间非常小的嵌入式应用。注意:如果要编译的项目引用 C++函数,那么由于 C++与 MicroLib 不兼容,所以此时不使用 MicroLib。

图 3.44 Target 标签页

Output 标签页用于设置输出的文件名与保存的目录，如图 3.45 所示。选中 Debug Information 选项可以生成调试信息，取消选中此选项时，无法打断点调试。选中 Create HEX File 选项可生成单独烧写的 Hex 文件。选中 Browse Infomation 选项可以生成浏览信息，可以在 Keil 中索引函数或变量的定义、调用等，没有这个信息就无法直接定位函数所在位置；取消选中该选项会大大加快文件的编译速度。当需要封装模块或打包静态库时，可以选中 Create Library 单选按钮，该选项与 Create Executable 互斥，会生成.lib 文件而不是完整的.axf 文件。该选项可用于提供复用的软件包使用，一般不勾选。选中 Create Batch File 选项即可在编译后生成.bat 格式的编译执行脚本，利用此脚本，可在不打开 Keil 工程的情况下只执行编译执行脚本即可编译工程。

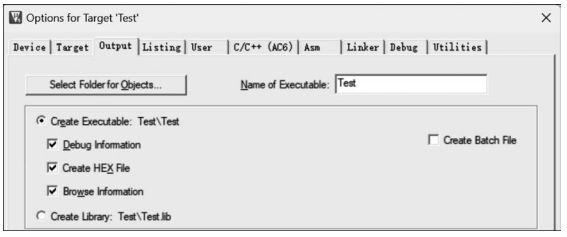

图 3.45　Output 标签页

Listing 标签页可以通过 Select Folder for Listings 按钮指定.lst 文件的存放目录，避免在编译过程中生成的.lst 临时文件杂乱无章，如图 3.46 所示。还可设置是否生成.map 等文件。

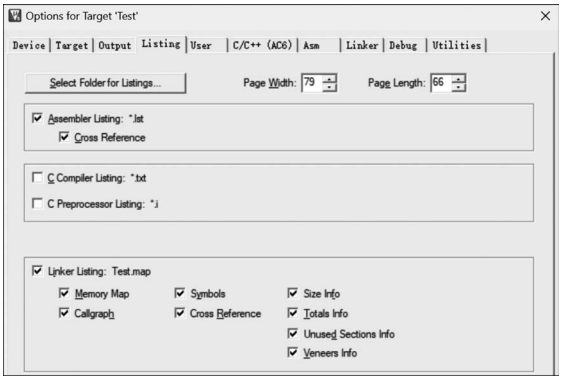

图 3.46　Listing 标签页

User 标签页可以指定编译前和编译后执行的用户程序。选中 Beep When Complete 选项时在编译完成后扬声器会响一下以进行提示。

C/C++(AC6)标签页用于设置编译 C 语言时的预定义、编译选项、头文件目录、编译警告等，如图 3.47 所示。

Define 中的“USE_HAL_DRIVER,STM32U575xx”是预编译宏，表示使用 HAL 库、STM32U575xx 系列单片机，对应程序中的预编译判断。如果有其他的预编译宏，则可以用逗号分开。

Optimization 可用于设置优化等级。当项目工程较大，想要节省芯片存储空间时，可以考虑提升优化等级。ST 的芯片这里有 0～3 共 4 个等级可选。需要注意的是，优化等级越

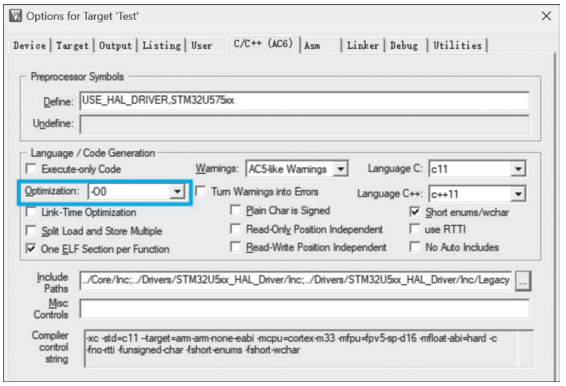

图 3.47　C/C++(AC6)标签页

高,在调试模式中能加入的断点位置越少。因此,这里选择-O0 模式,不进行优化。

Include Paths 用于指定头文件的搜索路径。

Misc Controls 用于设置是否忽略警告。如果在 MDK 中需要忽略某一个具体的警告,只需在 Misc Controls 中添加“--diag_suppress=< num >”就可以了,num 就是 Keil 中的警告代码。例如,在工程中需要忽略../Core/Src/main.c(88): warning: #177-D: variable "i" was declared but never referenced 这个警告,只需添加 --diag_suppress=177 即可。

Asm 标签页用于设置汇编编译时的选项。

Linker 标签页用于配置连接时的选项,如图 3.48 所示。其中,Scatter File(分散加载文件)可以设置分散加载文件的位置。Scatter File 描述内存的布局和分配,指定程序代码、数据和堆栈等的位置和大小,还可以指定在 Flash 存储器中的程序代码如何被分区和排列。当想在外部存储器中保存大量的数据时,可以利用分散加载文件将这些内容加载到外部存储中,避免内部存储不够用。

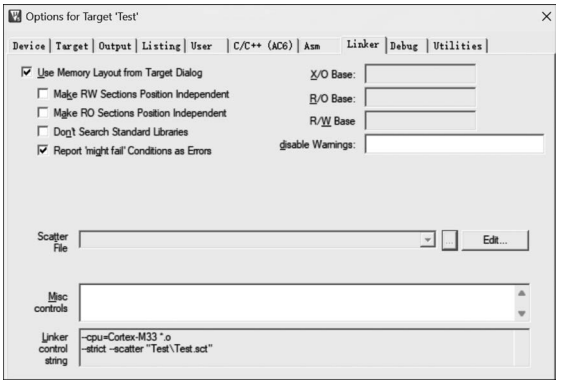

图 3.48　Linker 标签页

Debug 标签页用于设置软件模拟调试(左侧)和硬件调试(右侧),如图 3.49 所示。在硬件调试中,可以配置使用的调试器。

Use:选择调试器型号为 ST-Link Debugger。

Load Application at Startup:设置启动位置从启动文件开始加载。若不选中该选项,则在进入调试时,不会重新从初始启动位置开始执行,但需要手动添加.ini 文件,把.axf 的

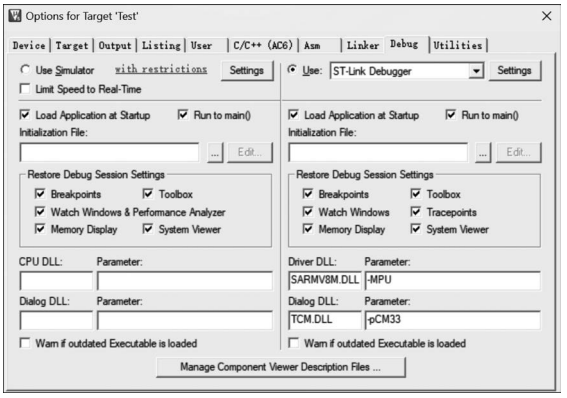

图 3.49　Debug 标签页

调试信息放到 Keil 里，否则进入调试时无法设置断点，也无法追踪到当前程序位置。

Initlalization File：指定可以包含 Debug Commands、Debug Functions、调试器配置和设备初始化命令的文件。

Run to main()：进入 Debug 模式后运行到 main 函数后暂停。如果不选中该选项，则会在启动文件处暂停。

Restore Debug Session Settings：恢复调试会话设置。选中该选项后可以使用上一次调试过程中设置的 Breakpoints、Watch Windows、Memory Display、Toolbox 等。

Driver DLL：驱动动态库文件，Parameter 是其参数。

Dialog DLL：会话框动态库文件，Parameter 是其参数。

Utilities 标签页用于配置 Flash 烧写算法和配置如何处理生成的镜像文件。

(3) 双击打开 main.c 文件，找到 int main(void)函数。代码中有很多 USER CODE BEGIN 和 USER CODE END 的标记。用户代码需要写在这些标记之内，才能在 STM32CubeMX 中更改配置后重新生成代码时不被覆盖。在 USER CODE BEGIN 1 处定义一个变量，然后在主循环中对这个变量+1，如下所示：

```
int main(void)
{
  /* USER CODE BEGIN 1 */
    int i = 0;
  /* USER CODE END 1 */
  /* MCU Configuration--------------------------------------------------------*/
  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();
  /* Configure the system clock */
  SystemClock_Config();
  /* Configure the System Power */
  SystemPower_Config();
  /* USER CODE BEGIN WHILE */
  while (1)
  {
     i++;
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
  }
```

（4）设置编译优化等级为-O0，如图 3.50 所示。

（5）修改完成后，单击左上角的 Rebuild 按钮，如图 3.51 所示。

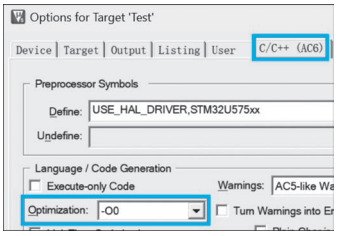

图 3.50　设置编译优化等级

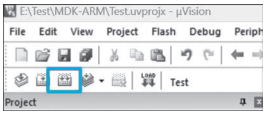

图 3.51　Rebuild 按钮所在位置

（6）在 Build Output 区域可以看到编译结果，如图 3.52 所示。在输出的信息中，Program Size 后面的 Code 代表代码段，存放程序的代码部分；RO-data 代表只读数据段，存放程序中定义的常量；RW-data 代表读写数据段，存放初始化为非 0 值的全局变量；ZI-data 代表 0 数据段，存放未初始化的全局变量及初始化为 0 的变量。其后跟的数字代表所占空间大小。

```
Build Output
compiling stm32u5xx_hal_flash.c...
compiling stm32u5xx_hal_flash_ex.c...
compiling stm32u5xx_hal_pwr_ex.c...
compiling stm32u5xx_hal_dma_ex.c...
compiling stm32u5xx_hal_rcc_ex.c...
linking...
Program Size: Code=5432 RO-data=680 RW-data=12 ZI-data=1644
FromELF: creating hex file...
"Test\Test.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:02
```

图 3.52　编译信息

这些信息会在工程目录下的 *.map 文件中详细说明，如图 3.53 所示。在该文件中，列举了每个编译后生成的.o 文件中不同数据所占的空间大小。

```
================================================================================

Image component sizes

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name

       126          0          0          0          0       2317   main.o
        68         28        568          0       1536        908   startup_stm32u575xx.o
       202          0          0          8          4      11715   stm32u5xx_hal.o
       162          0          0          0          0      14094   stm32u5xx_hal_cortex.o
        32          0          0          0          0       2160   stm32u5xx_hal_msp.o
       336          0          0          0          0      14698   stm32u5xx_hal_pwr_ex.o
      3994         12          0          0          0      15463   stm32u5xx_hal_rcc.o
        20          0          0          0          0       1002   stm32u5xx_it.o
        76          0         80          4          0       4230   system_stm32u5xx.o
    --------------------------------------------------------------------------------
      5046         40        680         12       1548      66587   Object Totals
         0          0         32          0          0          0   (incl. Generated)
        30          0          0          0          8          0   (incl. Padding)

    --------------------------------------------------------------------------------

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Member Name

         8          0          0          0          0         68   __main.o
         0          0          0          0          0          0   __rtentry.o
        12          0          0          0          0          0   __rtentry2.o
         6          0          0          0          0          0   __rtentry4.o
        52          8          0          0          0          0   __scatter.o
        26          0          0          0          0          0   __scatter_copy.o
        28          0          0          0          0          0   __scatter_zi.o
        18          0          0          0          0         80   exit.o
         6          0          0          0          0        152   heapauxi.o
         0          0          0          0          0          0   indicate_semi.o
```

图 3.53　map 文件内容

3.6.3　连接开发板调试程序

（1）将编译后的.hex文件烧写到芯片中。按照图3.54所示将开发板和计算机连接起来。其中，Type-C USB线用于给开发板供电；FS-DAP-LINK一端通过MiniUSB线连接到计算机，另一端通过排线和转接板连接到核心板上；核心板采用STM32U575（如果在创建工程时选择了其他型号的STM32，则可以选择对应的核心板）。

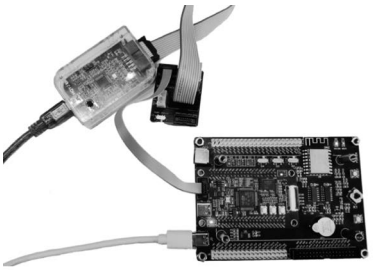

图3.54　开发板与DAP-LINK的连接

如果有ST-LINK，也可以将ST-LINK通过此种方式连接到开发板上。注意，要提前安装ST-LINK驱动并在设备管理器中正确识别到。

（2）连接好开发板和计算机并保证开发板供电后，在MDK中单击Options for Target按钮打开Debug标签页，依据使用的调试器选择CMSIS-DAP调试器或者ST-LINK调试器，如图3.55所示。

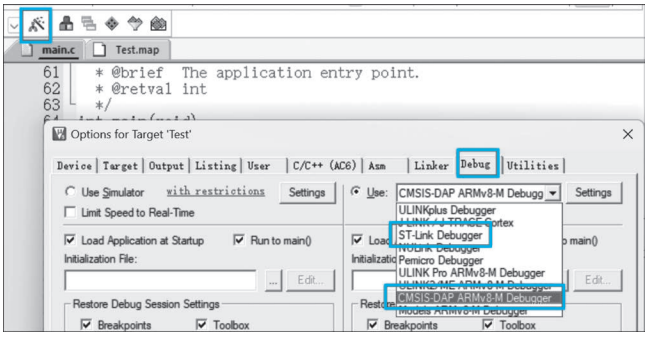

图3.55　调试器选择

（3）单击Settings按钮，在打开的页面中可以看到SW Device被正确识别。同时在Flash Download标签页里有相关的Flash烧写算法（若没有，则需要单击Add按钮添加），如图3.56所示。

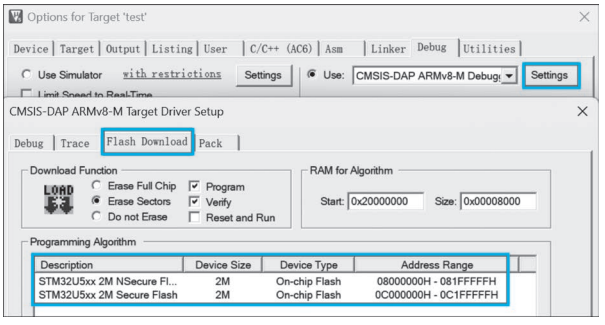

图3.56　Flash烧写算法设置页面

使用DAP-LINK时显示信息如图3.57所示。

使用ST-LINK时显示信息如图3.58所示。

（4）如图3.59所示，单击Debug按钮，弹出如图3.60所示的界面。

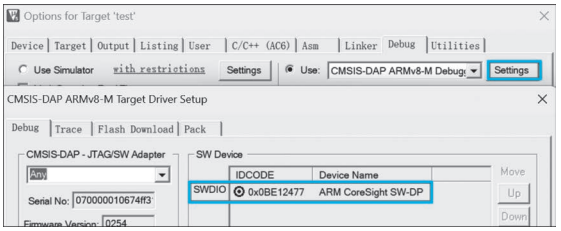

图 3.57　使用 DAP-LINK 时的识别信息

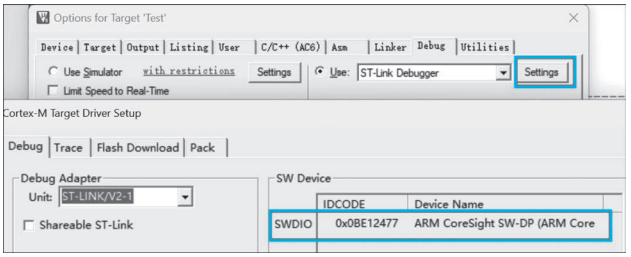

图 3.58　使用 ST-LINK 时的识别信息

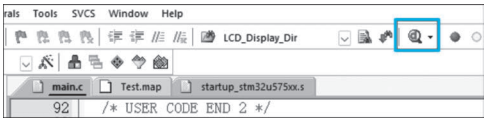

图 3.59　Debug 按钮所在位置

（5）在如图 3.60 所示中，左侧为寄存器列表，显示当前各个寄存器的值。右上侧的 Disassembly 为当前要执行的汇编指令。中间的代码界面由箭头指示下面要执行的代码。在 i++这个语句左侧深灰色位置单击，插入一个断点（如果无法插入断点，说明优化级别太高，需要调整优化级别为-O0，重新单击 Rebuild 按钮编译）。右下侧的 Call Stack 为相关函数、变量的地址、值和类型。

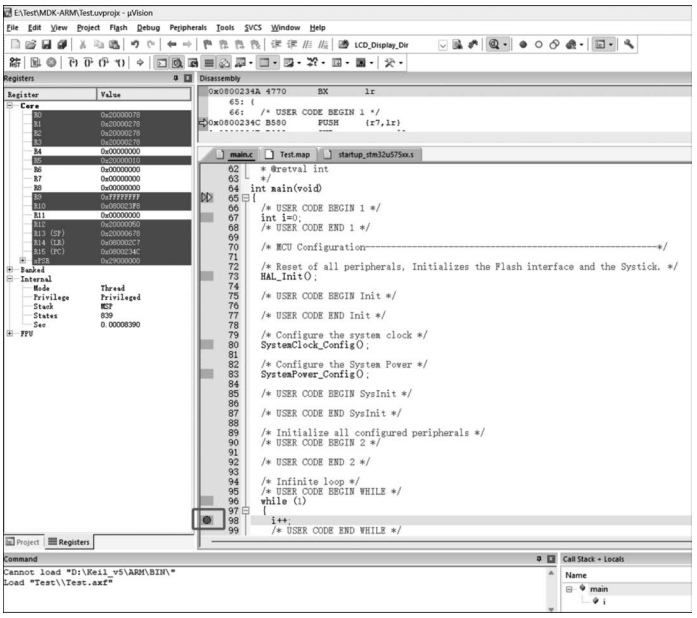

图 3.60　Debug 界面

在如图 3.61 所示的工具栏中，几个按钮分别为 Reset（让程序复位到起始位置）、Run（正常运行，但会在断点处暂停）、Stop（在当前运行的位置处暂停）、Step（单步执行，遇到函数时会跳进函数内）、Step Over（执行一行代码，不会跳进函数内）、Step Out（执行完毕当前函数后返回到上一层函数）、Run to Cursor Line（运行到光标所在行）、Show Next Statement（跳转到程序暂停所在行）。

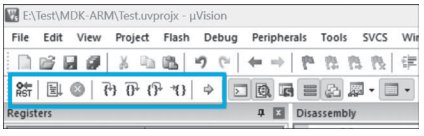

图 3.61　Debug 模式下的工具栏按钮

（6）单击两次 Step 按钮 {}，让程序执行完 i=0。可以看到此时箭头位于 HAL_Init() 函数前，说明即将执行该函数，同时右下角的变量 i 显示为 0，如图 3.62 所示。

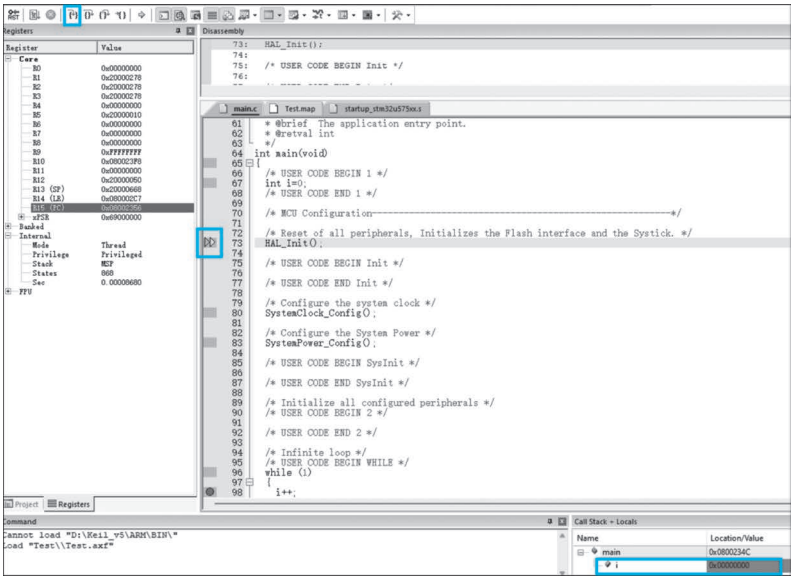

图 3.62　单击两次 Step 按钮后程序所在位置

（7）继续单击 Step 按钮 {}，程序进入 HAL_Init()函数内，如图 3.63 所示。

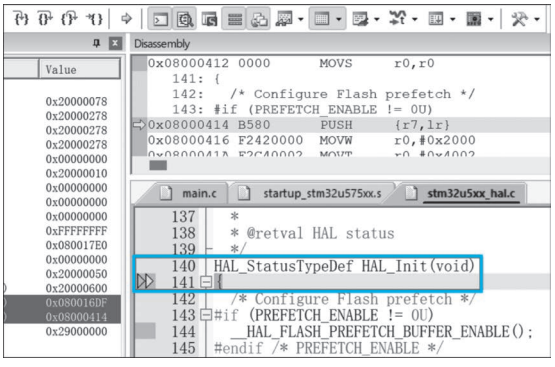

图 3.63　程序进入 HAL_Init()函数

（8）单击 Step Out 按钮 {}，让程序执行完本函数，并返回上一层函数。此时可以观察到箭头位于即将执行的 SystemClock_Config()函数前，如图 3.64 所示。

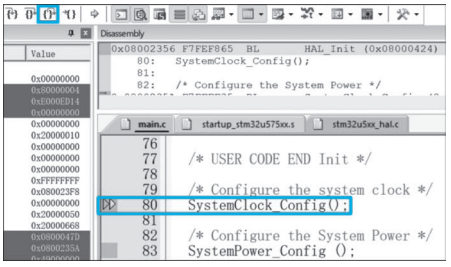

图 3.64　即将执行 SystemClock_Config()函数

（9）单击 Step Over 按钮 {}，即可执行完毕当前的行所代表的内容，光标移动到下一条代码上。

（10）单击 Run 按钮全速运行，会发现程序会在断点处暂停，如图 3.65 所示。因为还没有执行 i++，所以此时变量 i 的值仍然为 0。

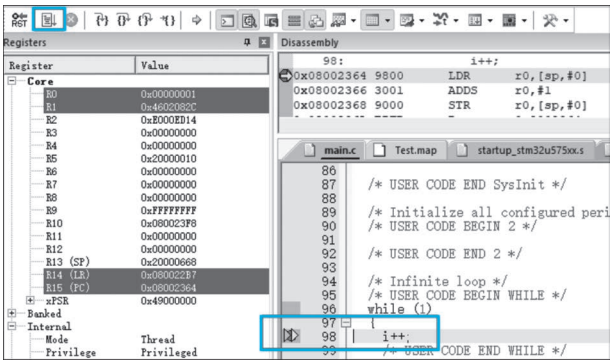

图 3.65　程序在断点处暂停

（11）继续单击 Run 按钮，让其在主循环中运行一圈后再次在断点处暂停，会发现变量 i 的值变为 1，如图 3.66 所示。

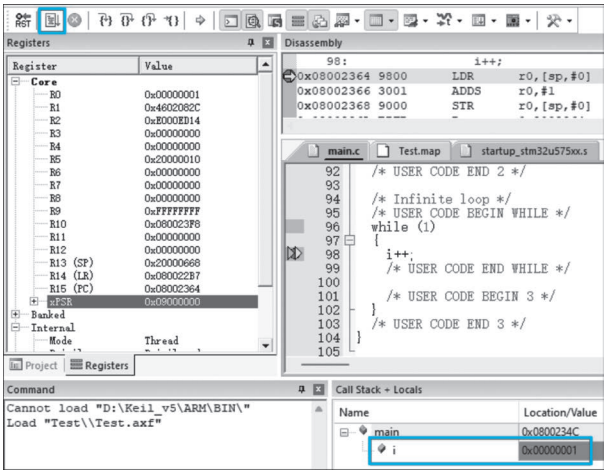

图 3.66　变量 i 的值变为 1

(12) 单击 Reset 按钮，程序会复位到代码运行的初始位置，如图 3.67 所示。

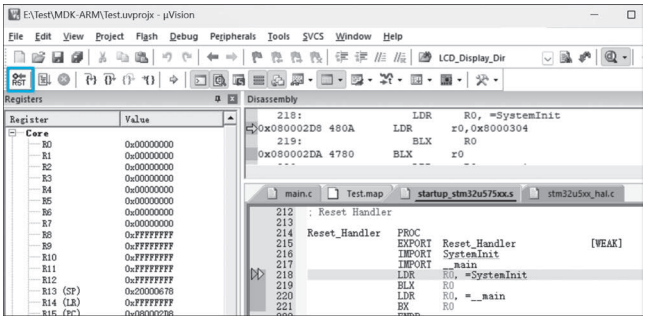

图 3.67　程序复位到代码运行的初始位置

3.7　main()函数之前的启动流程


视频讲解

在调试过程中，我们会发现程序从 main()函数中开始执行。实际上，在运行 main()函数之前，芯片会依据一些程序进行系统的初始化，然后才会调到 main()函数处执行。下面解释一下这个过程。

一般来说，STM32 处理器复位以后的工作步骤如图 3.68 所示(注意，此处说明的是典型情况，具体情况需要依据不同架构而定，特别是地址映射和 BOOT 引脚的配置部分)。

软复位是指通过编程方式在程序中触发的复位方式，如看门狗、寄存器写入等。硬复位是通过物理方式触发的复位方式，通常是通过将复位引脚(例如，NRST)拉低来实现的。无论是软复位还是硬复位，当复位发生时，STM32 都会通过 BOOT0 和 BOOT1 两个引脚(STM32U5 系列只有一个 BOOT0 引脚)的电平来判断启动程序加载到了哪里，即 0x00000000 这个地址实际代表哪个地址。

BOOT0＝0，BOOT1＝X：从主 Flash 内存启动，实际地址为 0x08000000。这是最常见的配置，用于正常的应用程序启动。

BOOT0＝1，BOOT1＝1：从 SRAM 启动，实际地址为 0x20000000 这需要在链接时由分散加载文件 *.sct 分配程序到 SRAM。

BOOT0＝1，BOOT1＝0：从系统存储器启动，实际地址为 0x1FFFF000，这是一段特殊的空间，通常包含出厂时设置的代码，为 ISP(In System Program)提供支持。它允许用户通过外部接口(如串行接口)来加载新的固件或进行其他调试和维护操作。

将程序加载到 Flash 或 SRAM 时，会执行启动文件 startup_stm32 **** xx.s(通过分散加载文件设置启动文件位于哪里)。在该文件中，首先会初始化堆栈空间，并将栈顶地址分配到最开始的 4 字节空间 0x00000000～0x00000003(如果程序烧进了 Flash，则映射到 0x08000000～0x08000003)。然后分配每个中断向量的地址。第一个中断向量是 Reset_Handler，也就是复位后要执行的程序，它被分配到栈顶地址后面的位置，即 0x00000004。在 Reset_Handler 中，会跳到 C 语言编写的初始化代码 SystemInit 中，然后再通过 __main 跳转到另一部分初始化代码，最终进入用户的 main()函数。

下面再用如图 3.69 所示的存储结构图来解释这个过程。该存储结构图为简化版，实际情况需要查看芯片手册的说明。

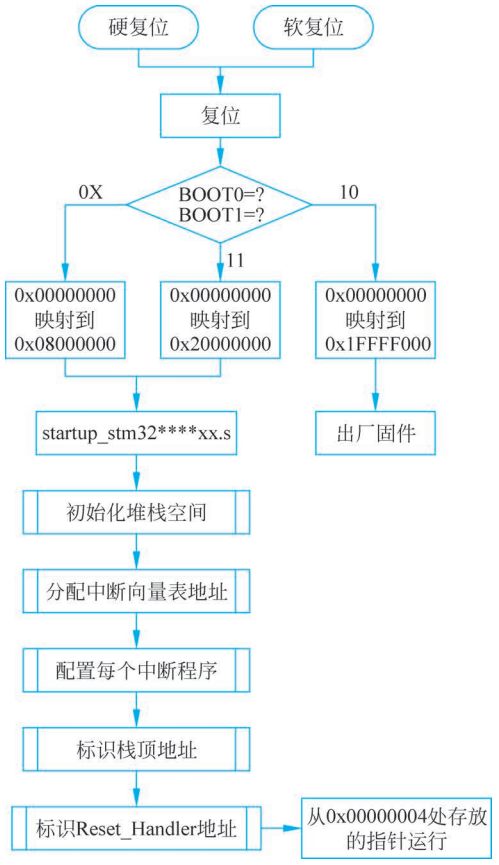

图 3.68 STM32 处理器复位以后的工作步骤

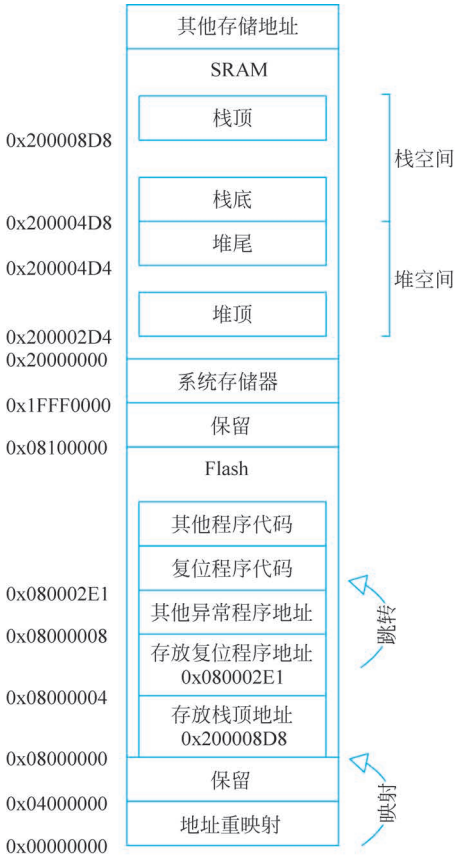

图 3.69 STM32 的存储结构图

当 BOOT0＝0，BOOT1＝X 时，0x00000000 映射到 0x08000000，于是系统会到此地址寻找栈顶地址是多少。根据分配的栈空间和堆空间大小，系统可能将 0x200008D8 作为栈顶地址，向下生长 0x400 字节的地址，紧接着是 0x200 字节大小的堆空间。所以 0x08000000 处存放的是栈顶地址 0x200008D8。

系统取出栈顶地址后，紧接着在 0x08000004 处取出程序地址。程序地址位于 0x80002E1，于是系统跳转到此处运行程序。

在调试模式中，可以在 Memory 窗口搜索到对应地址的数据，如图 3.70 所示。

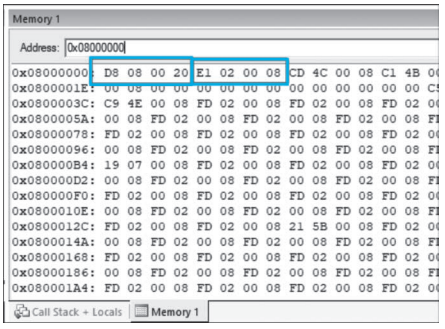

图 3.70 调试模式中的 Memory 窗口

启动代码 * .s 文件的作用

startup_stm32 **** xx.s 文件中的代码执行于 main()函数之前，主要用于设置初始堆栈指针(SP)、设置初始程序计数器(PC)等于 Reset_Handler、在向量表中为每个异常/中断设置相应的服务程序地址、跳转到 C 库中的__main(最终调用 main())。下面看一下每部分的实现方式。

在 * .s 文件中，首先设置了堆栈的空间大小，并开辟了一段连续的空间，分配了堆栈的地址。EQU 伪指令用于为程序中的常量、标号等定义一个等效的字符名称，类似于 C 语言的 # define。AREA 命令指示汇编程序汇编一个新的代码段或数据段，它指示汇编器在内存中为该区域分配空间，并可以指定该区域的属性，例如，代码区、数据区、只读区等。

```
; Amount of memory (in bytes) allocated for Stack
; Tailor this value to your application needs
; <h> Stack Configuration
; <o> Stack Size (in Bytes) <0x0 - 0xFFFFFFFF:8>
; </h>
Stack_Size      EQU         0x400               ;定义栈空间大小为 0x400(1K)
                AREA        STACK, NOINIT, READWRITE, ALIGN = 3
                ;开辟一段可读可写的数据空间,段名为 STACK,按照 8 字节对齐
                Stack_Mem   SPACE   Stack_Size   ;分配 Stack_Size 大小的连续内存空间
                __initial_sp                     ;栈顶地址

                ; <h> Heap Configuration
                ; <o>  Heap Size (in Bytes) <0x0 - 0xFFFFFFFF:8>
                ; </h>
                Heap_Size   EQU     0x200        ;设置堆的大小
                            AREA    HEAP, NOINIT, READWRITE, ALIGN = 3
                __heap_base                      ;表示堆的开始地址
                Heap_Mem    SPACE   Heap_Size    ;分配 Heap_Size 大小的连续内存空间
                __heap_limit                     ;表示堆的结束地址
```

使用 PRESERVE8 确保了生成的可执行文件中的数据按照 8 字节边界对齐，这符合 ARM Cortex-M 等处理器的要求，并有助于优化程序的性能和稳定性。使用 THUMB 指令告诉编译器和链接器生成 Thumb 模式下的代码。Thumb 模式是 ARM 架构中的一种指令集，它通过使用更紧凑的指令编码来提高代码密度，并且通常可以提高代码执行效率。Thumb 模式下的指令通常是 16 位宽度，而 ARM 模式下的指令是 32 位宽度。在 STM32 中，通常使用 Thumb 模式来编写启动代码和处理中断服务程序等关键部分，以节省内存空间并提高执行效率。因此，在启动文件中使用 THUMB 指令可以确保生成的代码在 Thumb 模式下运行。需要注意的是，可以通过特定的切换指令在 Thumb 模式和 ARM 模式之间进行转换，因此即使使用 Thumb 模式编写了启动文件，系统也可以在需要时从 Thumb 模式切换到 ARM 模式，反之亦然。

```
PRESERVE8                   ;指定当前文件所占空间按照 8 字节对齐
THUMB                       ;表示后面的指令兼容 THUMB 指令集
```

然后定义了一个中断服务函数向量表。中断向量表构建了中断源的识别标志，可用来形成相应的中断服务程序的入口地址或存放中断服务程序的首地址。EXPORT 伪指令用

于在程序中声明一个全局的标号，该标号可在其他的文件中引用。DCD 是 Data Constant Doubleword 的缩写，该指令用于定义一个或多个双字(32 位)的数据，并将这些数据初始化到内存中的指定位置。下列代码用 DCD 为每个中断服务程序分配了 4 字节的空间，该空间存放对应程序的入口地址。

```
; Vector Table Mapped to Address 0 at Reset
                AREA  RESET, DATA, READONLY     ;定义一块数据段,只读,名为 RESET
                EXPORT__Vectors                 ;连续空间的开始地址
                EXPORT__Vectors_End             ;连续空间的结束地址
                EXPORT__Vectors_Size            ;连续空间的大小

__Vectors       DCD   __initial_sp              ; Top of Stack
                DCD   Reset_Handler             ; Reset Handler
                DCD   NMI_Handler               ; NMI Handler
                DCD   HardFault_Handler         ; Hard Fault Handler
                DCD   MemManage_Handler         ; MPU Fault Handler
                DCD   BusFault_Handler          ; Bus Fault Handler
                DCD   UsageFault_Handler        ; Usage Fault Handler
                DCD   SecureFault_Handler       ; Secure Fault Handler
                DCD   0                         ; Reserved
                DCD   0                         ; Reserved
                DCD   0                         ; Reserved
                DCD   SVC_Handler               ; SVCall Handler
                DCD   DebugMon_Handler          ; Debug Monitor Handler
                DCD   0                         ; Reserved
                DCD   PendSV_Handler            ; PendSV Handler
                DCD   SysTick_Handler           ; SysTick Handler
                ; External Interrupts
                DCD   WWDG_IRQHandler           ; Window WatchDog
                DCD   PVD_PVM_IRQHandler        ; PVD/PVM through EXTI Line detection Interrupt
                DCD   RTC_IRQHandler            ; RTC non - secure interrupt
                DCD   RTC_S_IRQHandler          ; RTC secure interrupt
                DCD   TAMP_IRQHandler           ; Tamper non - secure interrupt
                …
```

然后定义了一段只读的代码区域。

```
                AREA    |.text|, CODE, READONLY
; Reset Handler
Reset_Handler    PROC
                EXPORT  Reset_Handler             [WEAK]
                IMPORT  SystemInit
                IMPORT  __main
                LDR     R0, = SystemInit
                BLX     R0
                LDR     R0, = __main
                BX      R0
                ENDP
; Dummy Exception Handlers (infinite loops which can be modified)
NMI_Handler\
                PROC
                EXPORT  NMI_Handler               [WEAK]
```

```
                B       .
                ENDP
HardFault_Handler\
                PROC
                EXPORT  HardFault_Handler         [WEAK]
                B       .
                ENDP
MemManage_Handler\
                PROC
                EXPORT  MemManage_Handler         [WEAK]
                B       .
                ENDP
BusFault_Handler\
                PROC
                EXPORT  BusFault_Handler          [WEAK]
                B       .
                ENDP
            …
```

在这段代码中，Reset_Handler 过程是处理处理器复位的代码。在 ARM Cortex-M 处理器中，复位后会跳转到该处理器的复位向量处执行。首先，用 EXPORT 声明了 Reset_Handler 过程具有全局属性，以便其他文件可以引用它。其次，导入了 SystemInit 和__main，分别表示系统初始化函数和主函数。通过“LDR R0，=SystemInit”命令将 SystemInit 函数的地址加载到寄存器 R0 中，通过 BLX R0 调用 SystemInit 函数。再次，通过“LDR R0，=__main”命令将__main 函数的地址加载到寄存器 R0 中。最后，通过 BX R0 跳转到__main 函数执行主程序。

NMI_Handler、HardFault_Handler 等过程是处理异常情况的代码，例如，NMI 中断、硬件错误等。每个过程都是一个无限循环，其中只包含了一个无条件分支到当前指令地址的指令“B.”。这相当于一个空循环，可以在需要时修改成相应的处理代码。

总的来说，这段代码的作用是在处理器复位时执行系统初始化，并跳转到主程序，同时定义了一些异常处理函数，这些函数目前只是简单的无限循环，可以在实际应用中根据需要进行修改。

最后根据是否使用微型库 MicroLib 来初始化用户堆栈和堆。如果使用微型库，则声明相关符号以便链接器使用；如果未使用微型库，则定义初始化堆栈和堆的过程，并声明相应的符号。

```
;*------------------------------------------------------------------
; User Stack and Heap initialization
;*------------------------------------------------------------------
            IF      :DEF:__MICROLIB

            EXPORT  __initial_sp
            EXPORT  __heap_base
            EXPORT  __heap_limit

            ELSE

            IMPORT  __use_two_region_memory
```

```
                EXPORT  __user_initial_stackheap

__user_initial_stackheap PROC
                LDR     R0, =   Heap_Mem
                LDR     R1, =(Stack_Mem + Stack_Size)
                LDR     R2, = (Heap_Mem +   Heap_Size)
                LDR     R3, = Stack_Mem
                BX      LR
                ENDP

                ALIGN

                ENDIF

                END
```