深入浅出物联网操作系统 Mbed

Version 0.1

作者：吴　昊

QQ：　66322494

邮件：wuhao@yiweitech.net

目　　录

1 概览

2014 年，ARM 宣布了针对物联网低功耗设备的操作系统 Mbed OS。mbed OS 部分开源，其余部分控制在 ARM 手中，理由是为了确保操作系统不会碎片化。ARM 声称 Mbed OS 只需要 256kb 内存，它希望开发商能使用 Mbed 开发电池使用寿命长达数年的设备。Mbed OS 将免费提供给所有厂商使用。

Mbed 软件体系架构分为应用层、中间层和硬件层，Mbed 主要实现了中间层功能部分功能。

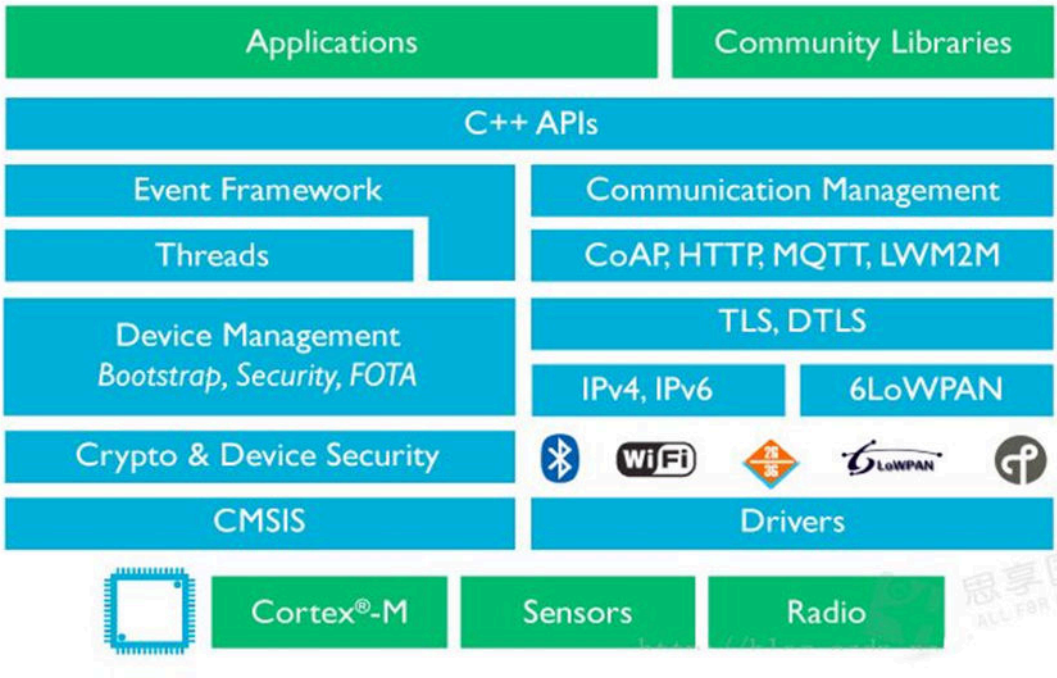


图 1-1：Mbed 系统架构图

Mbed 在国内应用比较少，其中主要几个原因：其一，Mbed 主要推荐在线开发，开发完成后下载编译后固件并运行，这种模式存在几种问题：无法在线调试硬件，国内网络环境无法访问部分国外网站，在线开发不利于知识产权保护，并且 Mbed 导出本地工程也存在一些问题：其二，国内嵌入式开发以 Keil 为主，而 Keil 对 C++支持比较弱，gcc 主要以命令模式进行开发，使用比较不友好。

本文主要讲述了基本 Visual Studio 2015（后面简称 VS2015）和 VisualGDB 全图形界面物联网操作系统 Mbed 本地化移植和使用方法。

2 Mbed 移植

要了解和使用一个操作系统最好的办法是使用,为了更好的了解 Mbed 需要移植，嵌入开发需要软件环境和硬件环境两方面。

2.1 移植需要的软硬件环境

2.1.1 软件环境

Windows 7 X64、Visual Studio 2015、VisualGDB、串口调试工具、Mbed OS 源码

2.1.2 硬件环境

STM32F429 开发板，CMSIS-DAP 调试器，串口转 USB 线

2.1.3 VisualGDB 概述

VisualGDB 和 Visual Studio 相结合，使得跨平台开发非常容易和简便。支持以下特性：

- 嵌入式 Barebone 系统和 IoT 模块
- C/C++ Linux 应用程序
- 原生 Android 应用程序及库
- Raspbery Pi 及其他的 Linux boards
- Linux 内核模块（需要单独的 VisualKernel 产品）
- 使用扩展 API，可以将你的设备及平台作为目标。

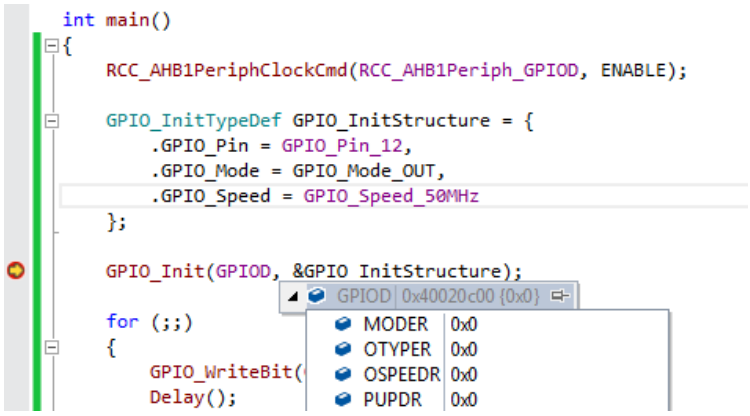
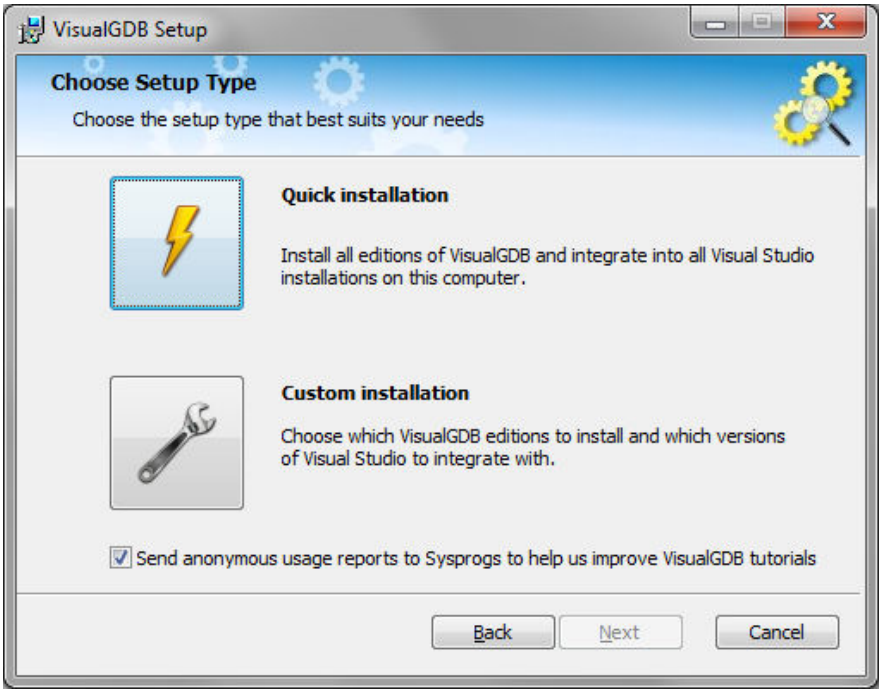


图 1-2：VisualGDB 调试 ARM 控制器

2.2 软件环境准备

2.2.1 安装 Visual Studio 2015。

2.2.2 安装 VisualGDB

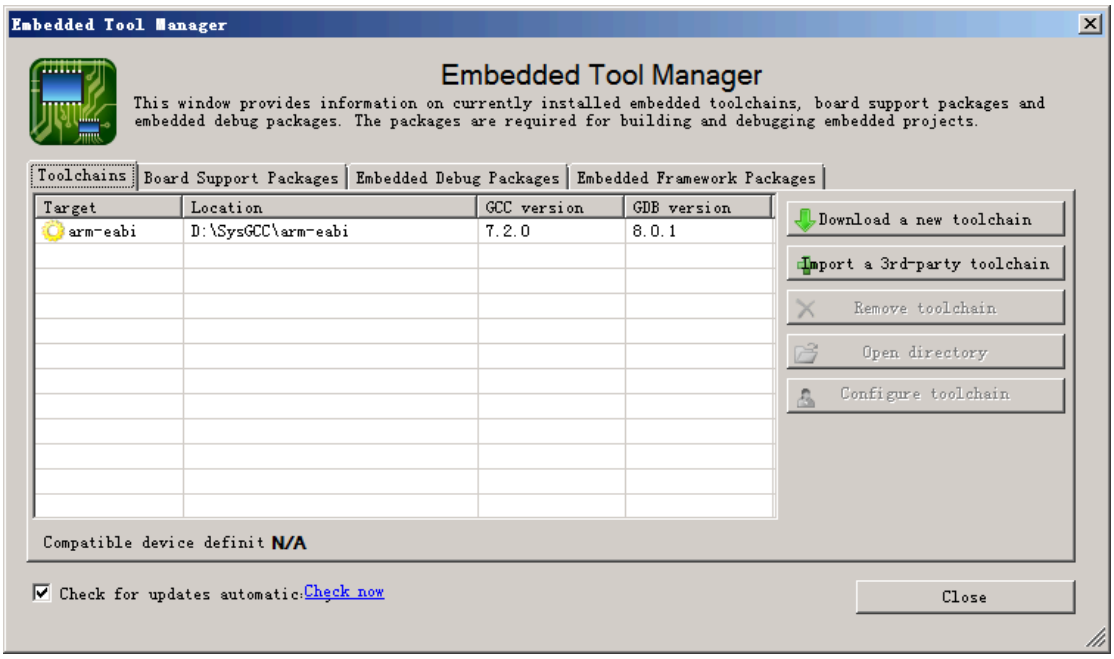


2.2.3 安装工具

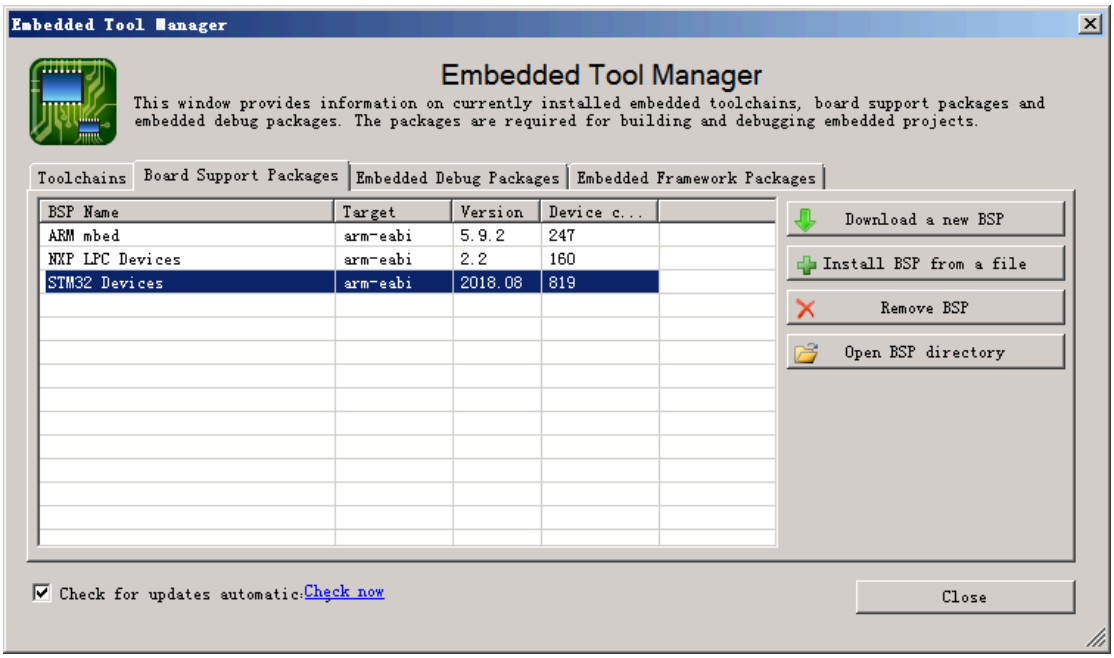
打开 VS2015，点“工具”-“Embedded Tools Manager”。


MbedTest - Microsoft Visual Studio
文件(F) 编辑(E) 视图(V) VAssistX 项目(P) 生成(B)
工具(T) Android 测试(S) 分析(N) 窗口(W) 帮助(H)
连接到数据库(D)...
连接到服务器(S)...
Linux Source Cache Manager
SSH Host Manager
Shared Folder Manager
Embedded Tools Manager
代码片段管理器(T)... Ctrl+K, Ctrl+B
选择工具箱项(X)...
NuGet 包管理器(N)
扩展和更新(U)...
创建 GUID(G)
错误查找(K)
PreEmptive Protection - Dotfuscator
Spy++(+)
Spy++ x64
WCF 服务配置编辑器(W)
外部工具(E)...
导入和导出设置(I)...
自定义(C)...
选项(O)...

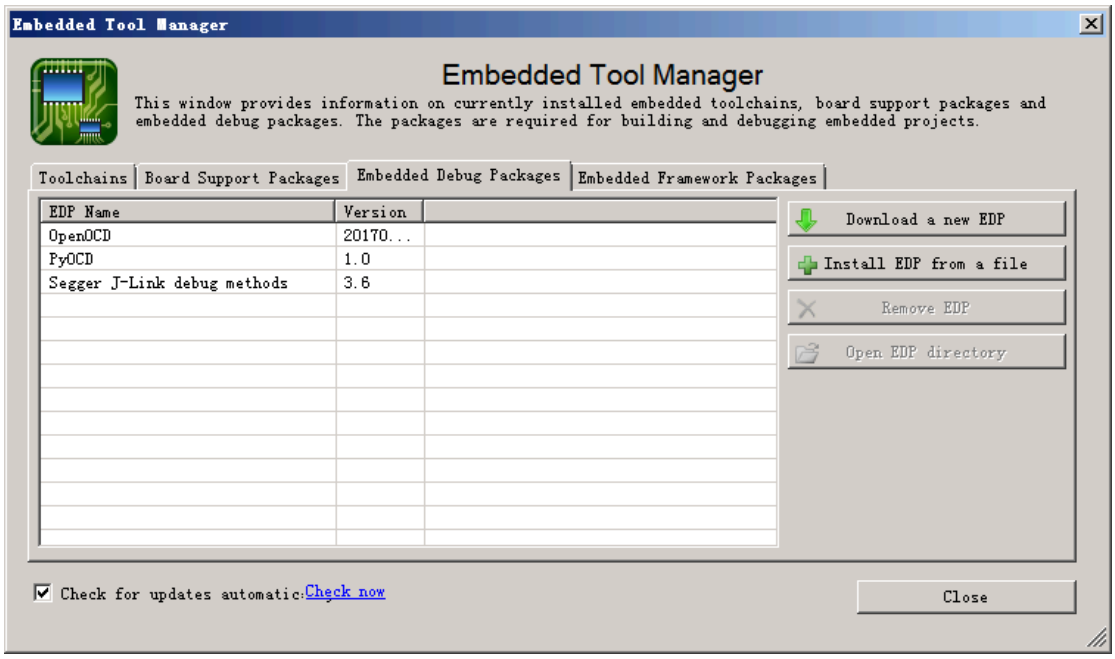
2.2.4 安装 GCC


Embedded Tool Manager
Embedded Tool Manager
This window provides information on currently installed embedded toolchains, board support packages and embedded debug packages. The packages are required for building and debugging embedded projects.
Toolchains
Board Support Packages
Embedded Debug Packages
Embedded Framework Packages
Target
Location
GCC version
GDB version
arm-eabi
D:\SysGCC\arm-eabi
7.2.0
8.0.1
Download a new toolchain
Import a 3rd-party toolchain
Remove toolchain
Open directory
Configure toolchain
Compatible device definit N/A
Check for updates automatic
Check now
Close

2.2.5 安装 STM32 BSP 包

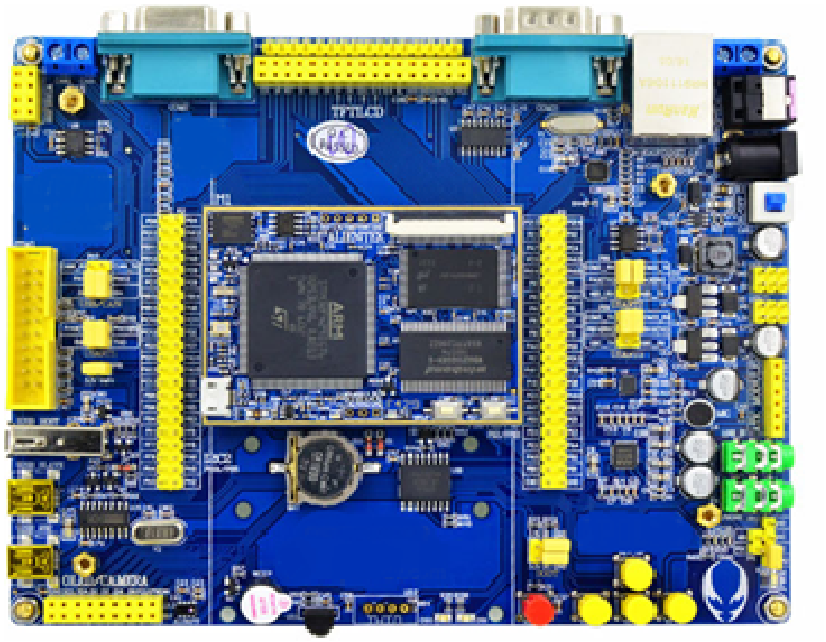


2.2.6 安装 OpenOCD 调试工具包

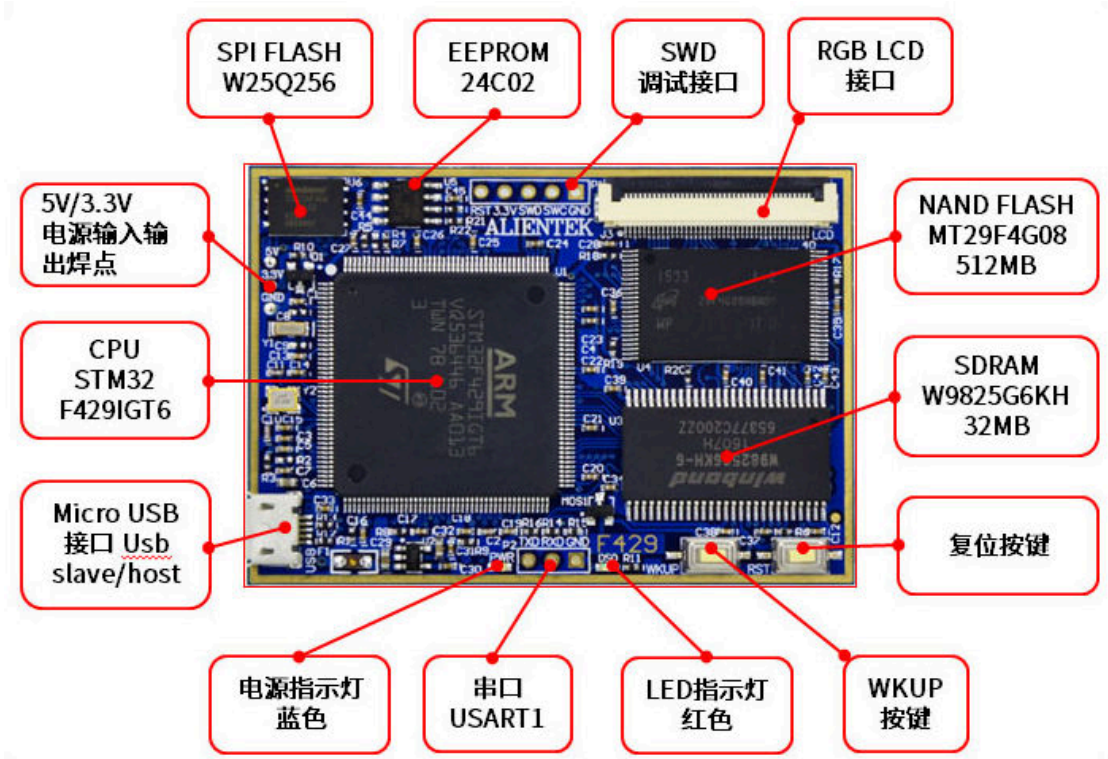


2.3 硬件环境准备

2.3.1 STM32F429 开发板



2.3.2 核心板资源简介



2.4 Mbed 源码结构

2.4.1 Mbed 重要目录

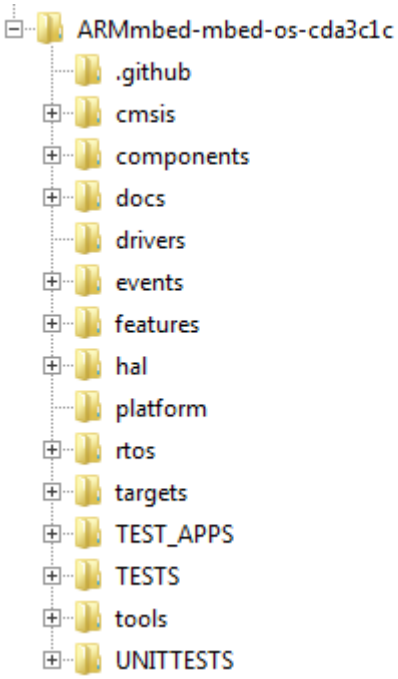
下载 Mbed 源代码：https://os.mbed.com/releases/，最新版本为 5.10.4 Mbed 源代码结构，我们主要关心与硬件相关：

MCU 无关层（MCU independent directories）：

- mbed/api：包含定义了实际的 mbed library API 的头文件
- mbed/common：源文件（mbed common sources）
- mbed/hal：包含了目标 MCU 需要实现的硬件抽象层接口（HAL API）
- mbed/rtos:包含嵌入式操作系统 RTX，后期将移植 FreeRTOS

MCU 相关层（MCU dependent directories）：

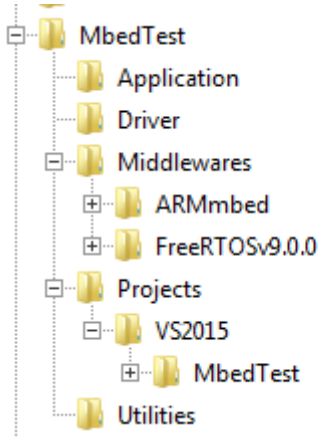
- mbed/cmsis：CMSIS-CORE 源文件
- mbed/targets：针对不同 MCU，实现 HAL 接口



2.5 工程目录准备

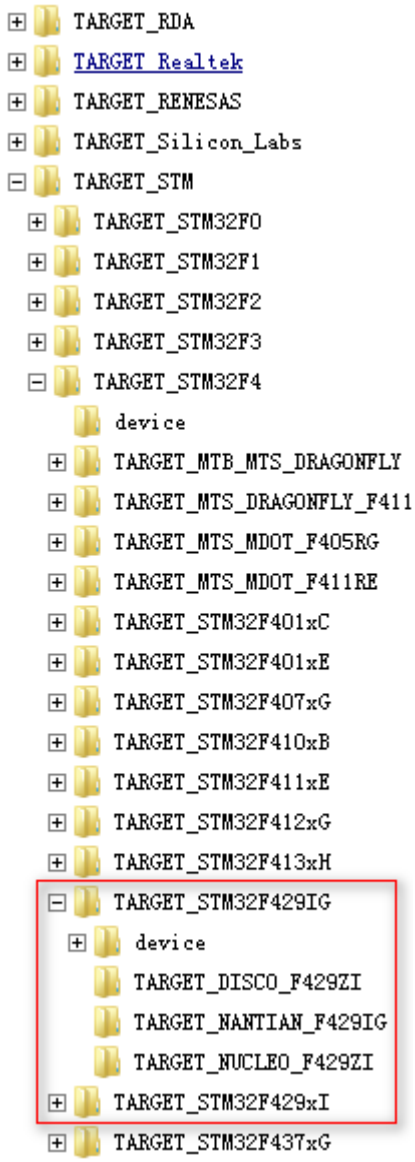
2.5.1 创建目录

Application—产品应用代码
Driver—外围芯片级驱动
Middleware—中间层软件，存放 Mbed 和 FreeRTOS 源码
Project—工程文件
Utilities—工具



2.5.2 创建目标平台目录

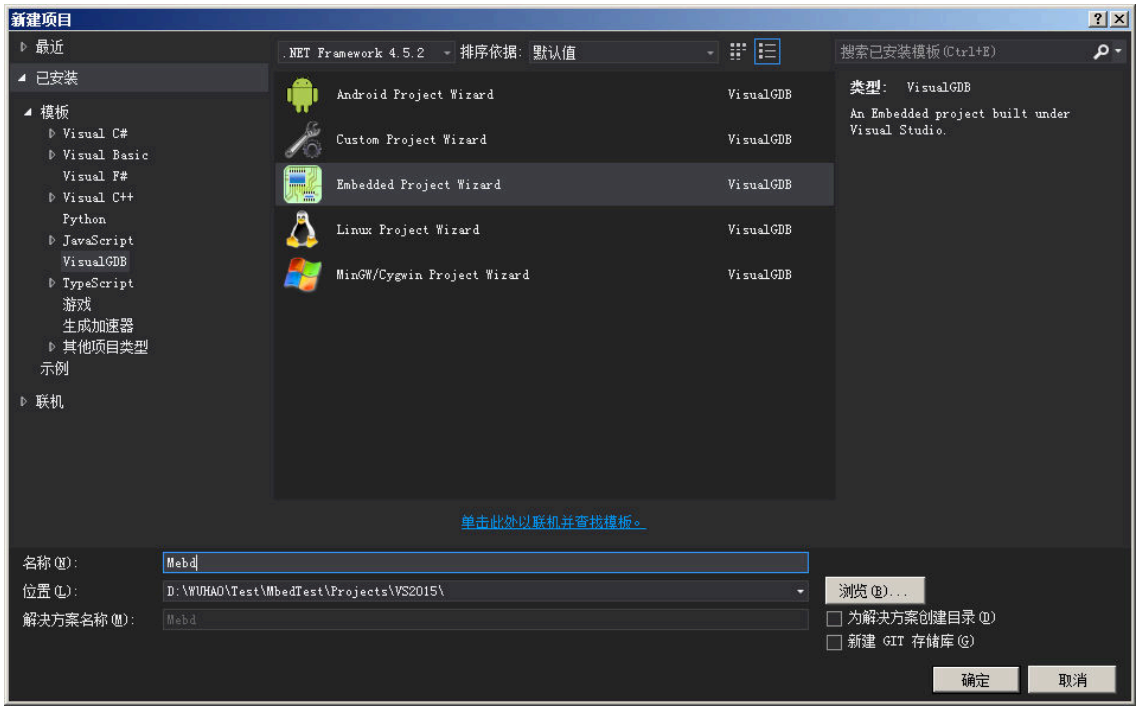
在 target 目录下包括很多 Mbed 现在支持的硬件平台，包括 Atmel、NXP、ST 等公司 ARM 系统控制器,在 TARGET_STM 目录包括 STM32F 系列和 STM32L 系统众多控制器，现需要把 Mbed 移植到 STM32F429IG 开发板中，找到 TARGET_STM32F4 目录，发现只有 TARGET_STM32F429xI 与目标比较接近，创建 TARGET_STM32F429IG（后简称平台目录）并从 TARGET_STM32F429xI 内容复制过来，并创建 TARGET_NANTIAN_F429IG 并从 TARGET_DISO_F429ZI 中内容复制过来。



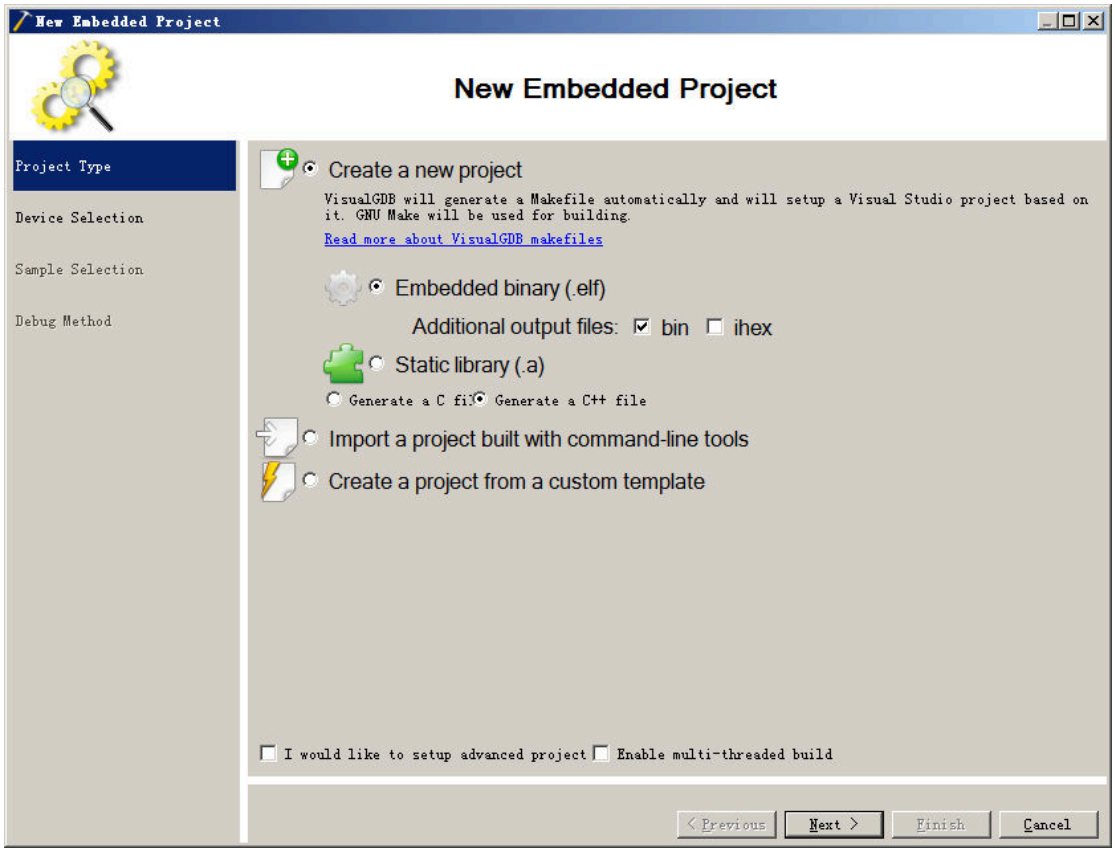
2.6 创建工程

2.6.1 选择项目类型

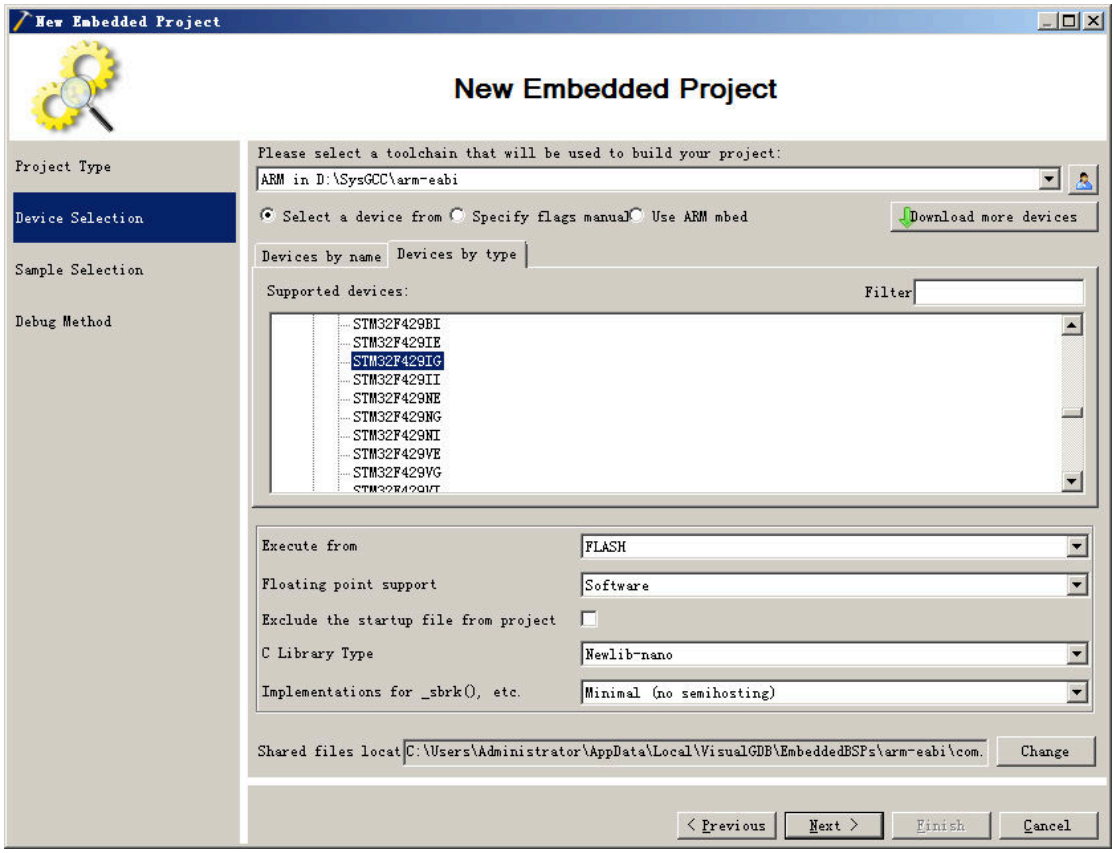
打开 VS2015，选择“File->New project”，然后选择“VisualGDB->Embedded Project Wizard”，并设定项目位置后点击“OK”。


新建项目
最近
已安装
模板
Visual C#
Visual Basic
Visual F#
Visual C++
Python
JavaScript
VisualGDB
TypeScript
游戏
生成加速器
其他项目类型
示例
联机
.NET Framework 4.5.2
排序依据: 默认值
搜索已安装模板(Ctrl+E)
Android Project Wizard VisualGDB
Custom Project Wizard VisualGDB
Embedded Project Wizard VisualGDB
Linux Project Wizard VisualGDB
MinGW/Cygwin Project Wizard VisualGDB
类型: VisualGDB
An Embedded project built under Visual Studio.
单击此处以联机并查找模板。
名称(N):
Mebd
位置(L):
D:\WUHAO\Test\MbedTest\Projects\VS2015\
解决方案名称(M):
Mebd
浏览(B)...
为解决方案创建目录(D)
新建 GIT 存储库(G)
确定
取消

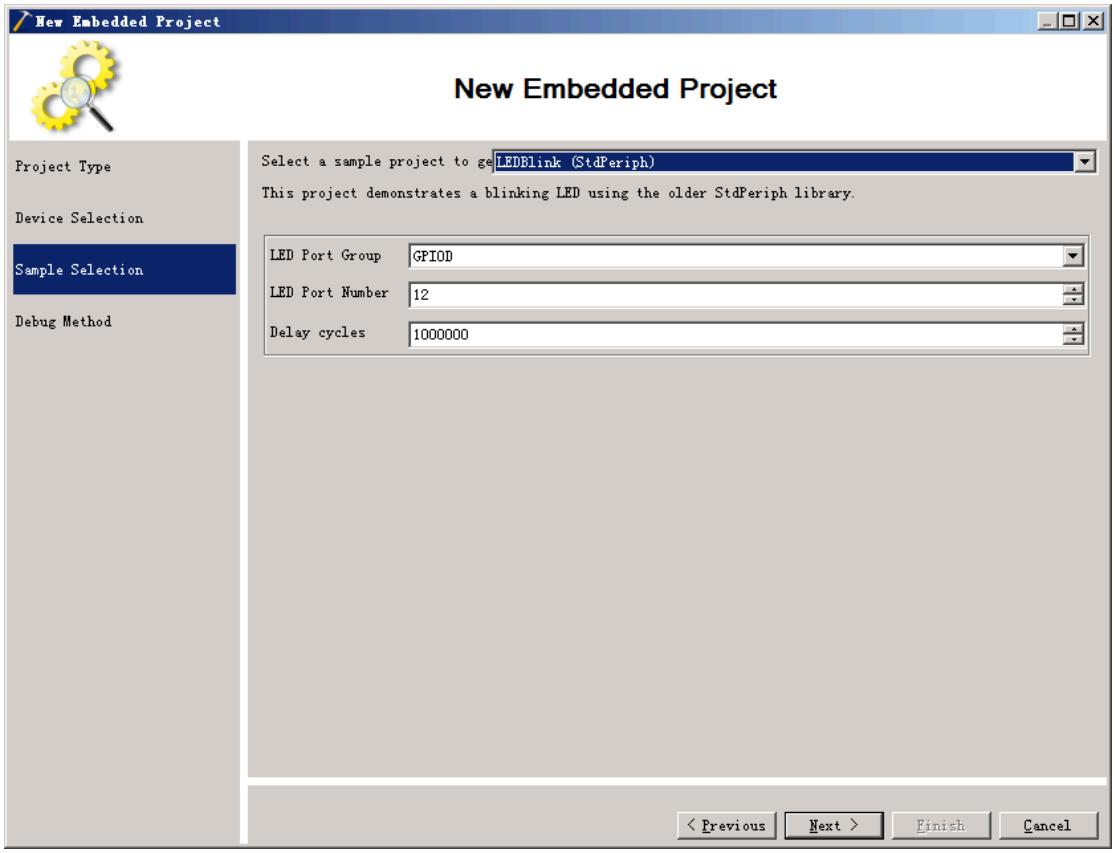
2.6.2 项目类型


New Embedded Project
New Embedded Project
Project Type
Device Selection
Sample Selection
Debug Method
Create a new project
VisualGDB will generate a Makefile automatically and will setup a Visual Studio project based on it. GNU Make will be used for building.
Read more about VisualGDB makefiles
Embedded binary (.elf)
Additional output files: bin ihex
Static library (.a)
Generate a C fi Generate a C++ file
Import a project built with command-line tools
Create a project from a custom template
I would like to setup advanced project
Enable multi-threaded build
< Previous
Next >
Finish
Cancel

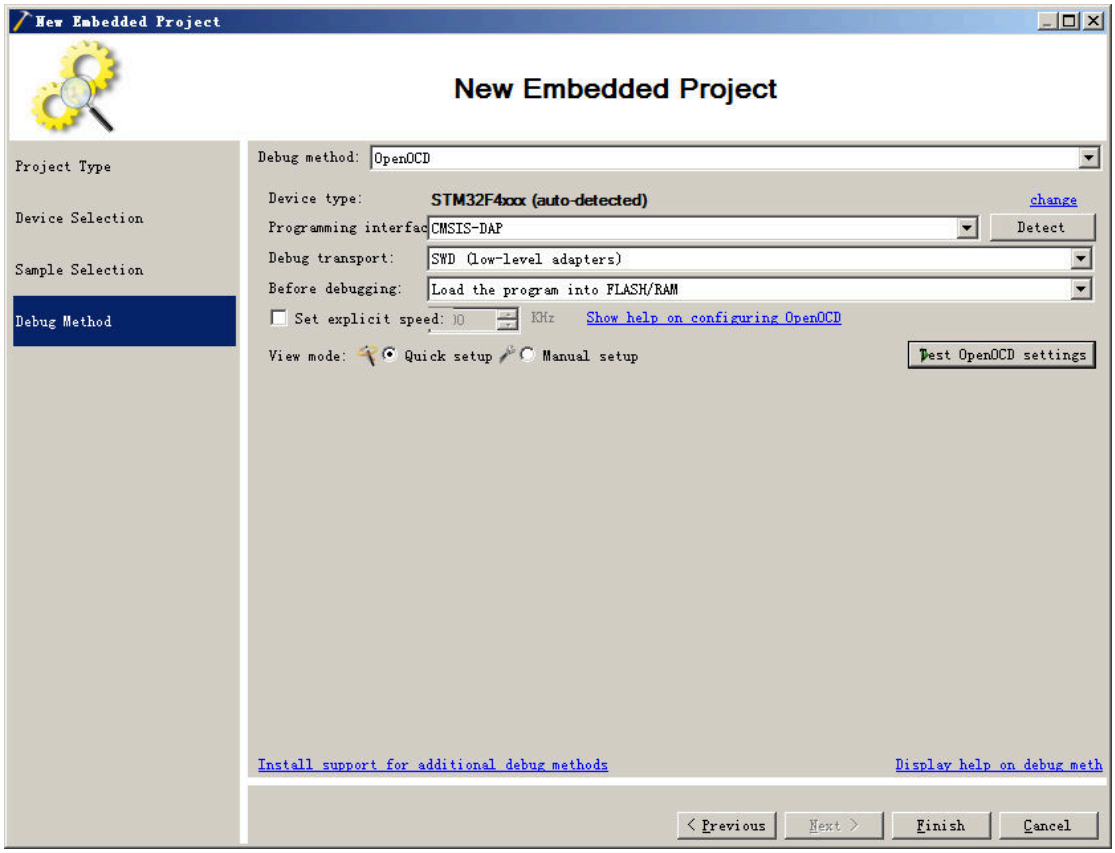
2.6.3 选择硬件平台


New Embedded Project
New Embedded Project
Project Type
Device Selection
Sample Selection
Debug Method
Please select a toolchain that will be used to build your project:
ARM in D:\SysGCC\arm-eabi
Select a device from
Specify flags manual
Use ARM mbed
Download more devices
Devices by name
Devices by type
Supported devices:
Filter
STM32F429BI
STM32F429IE
STM32F429IG
STM32F429II
STM32F429NE
STM32F429NG
STM32F429NI
STM32F429VE
STM32F429VG
Execute from
FLASH
Floating point support
Software
Exclude the startup file from project
C Library Type
Newlib-nano
Implementations for _sbrk(), etc.
Minimal (no semihosting)
Shared files locat
C:\Users\Administrator\AppData\Local\VisualGDB\EmbeddedBSPs\arm-eabi\com.
Change
< Previous
Next >
Finish
Cancel

2.6.4 选择开发库

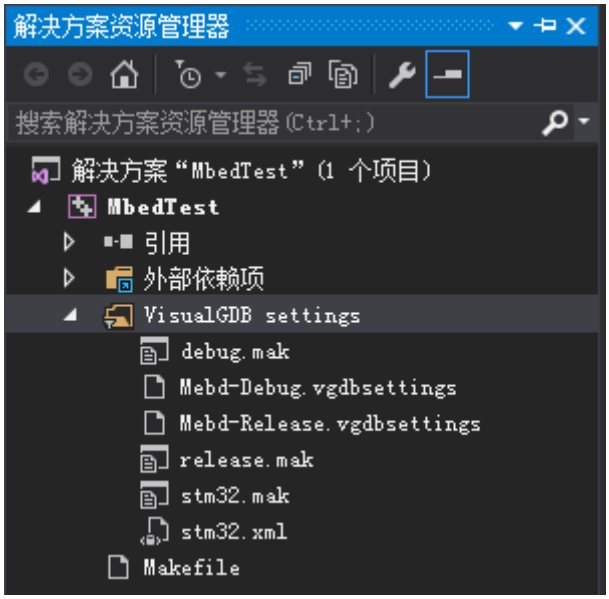

New Embedded Project
New Embedded Project
Project Type
Device Selection
Sample Selection
Debug Method
Select a sample project to ge
LEDBlink (StdPeriph)
This project demonstrates a blinking LED using the older StdPeriph library.
LED Port Group
GPIOD
LED Port Number
12
Delay cycles
1000000
< Previous
Next >
Finish
Cancel

2.6.5 选择调试方法

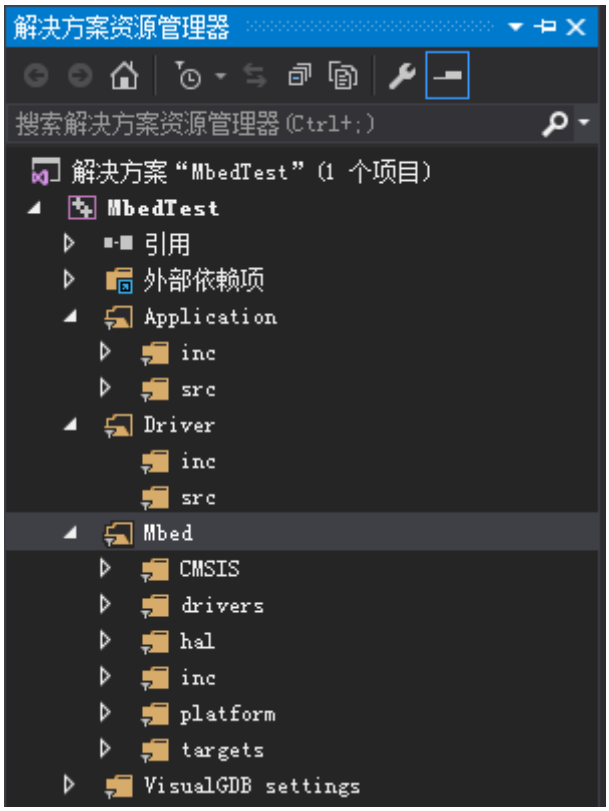


2.7 工程设置

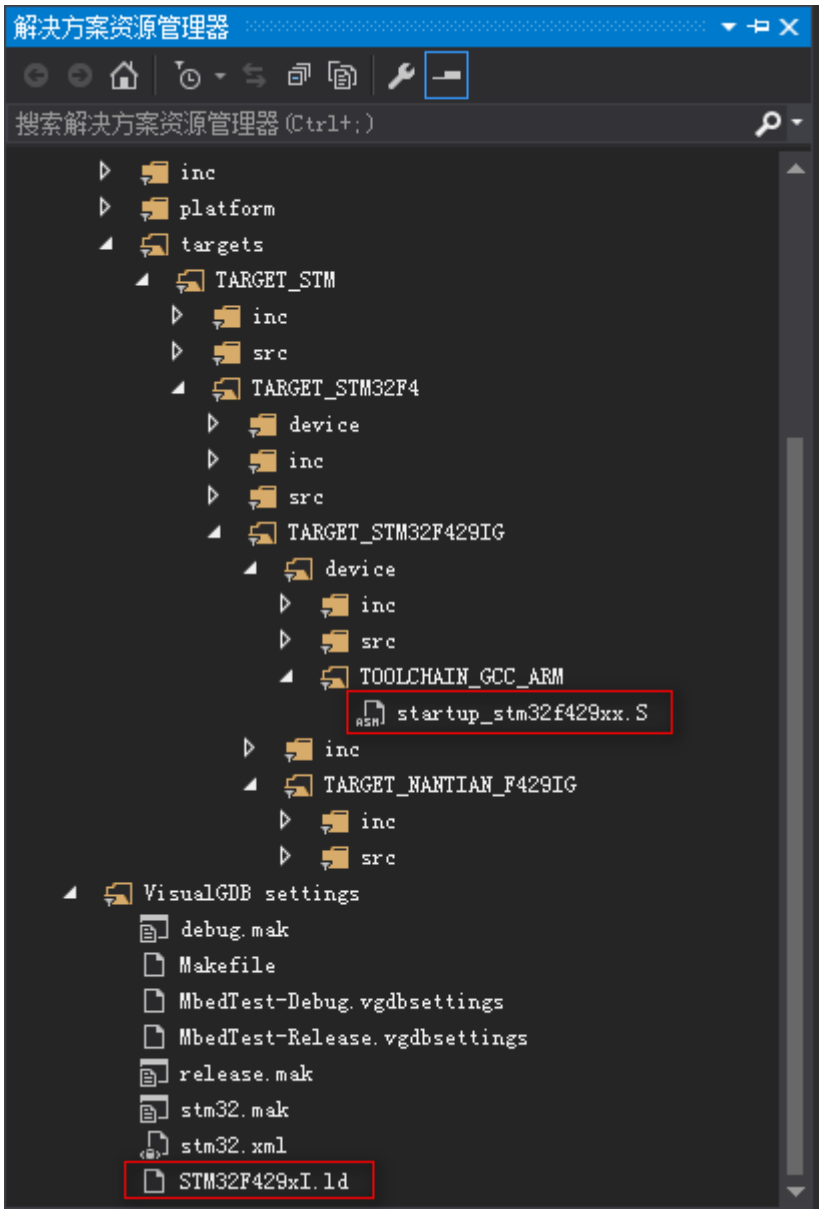
2.7.1 删除工程文件



2.7.2 增加项目目录

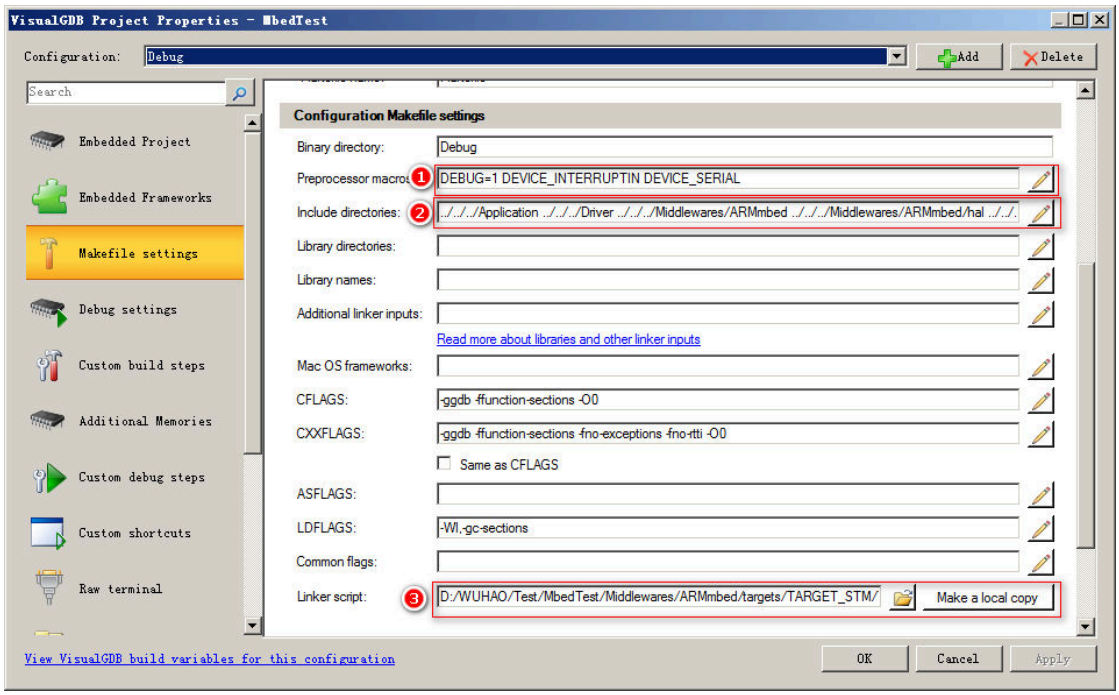

解决方案资源管理器
搜索解决方案资源管理器(Ctrl+;)
解决方案“MbedTest”(1 个项目)
MbedTest
引用
外部依赖项
Application
inc
src
Driver
inc
src
Mbed
CMSIS
drivers
hal
inc
platform
targets
VisualGDB settings

2.7.3 增加 Mbed 源文件

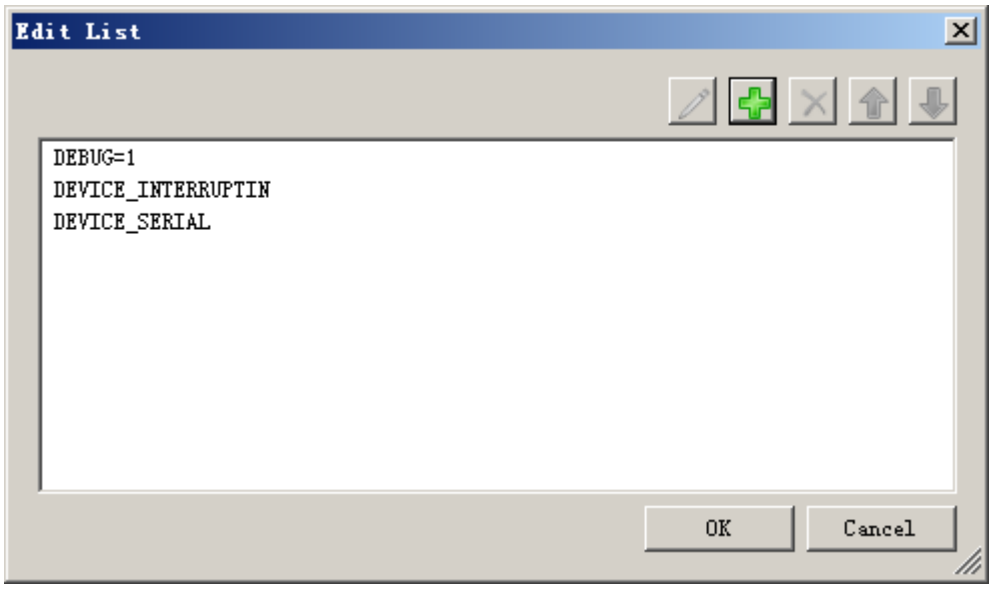


启动文件：startup_stm32f429xx.S，程序入口在此文件中。
链接脚本文件：STM32F429xI.ld，此文件是要告诉编译器怎样链接工程。

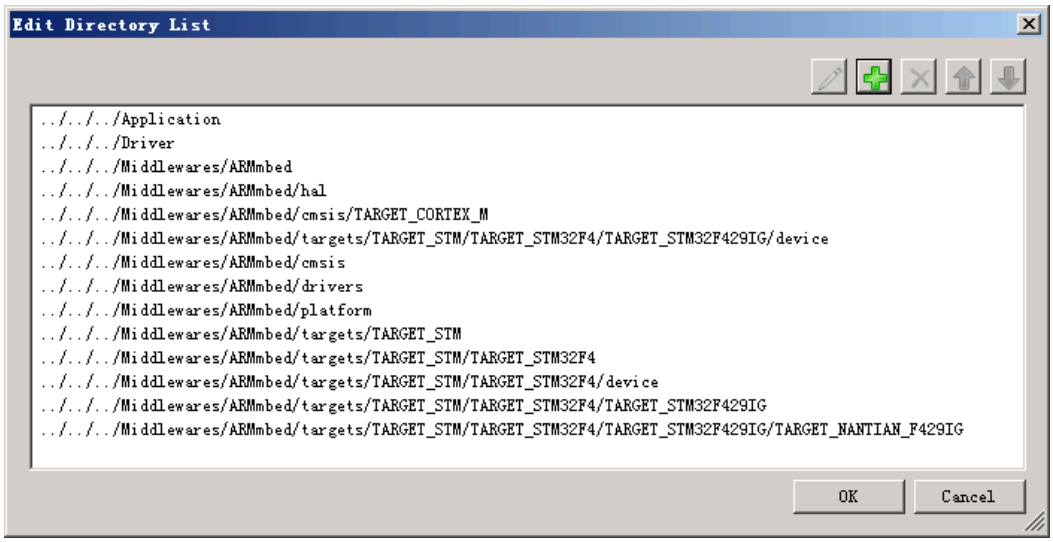
2.7.4 更改工程选项



- 增加 DEVICE_INTERRUPTIN DEVICE_SERIAL 这两个宏。



- 设置头文件目录



- 更改链接文件为 STM32F429xI.ld。
- 在 Application 文件夹增加主文件 main.cpp

```
#include "mbed.h"
void Delay_ms(uint32_t delay)
{
    uint32_t i;
    for (i=0;i<0xFFFFF;i++)
    {
    }
}
int main()
{
    printf("\n\n*** RTOS basic example ***\n");
    DigitalOut led1(LED1);
    while (true)
    {
        led1=!led1;
        Delay_ms(100);
    }
}
```

2.8 修改工程

2.8.1 增加硬件浮点数支持

更改 STM32.MAK 文件，-mfloat-abi=hard

```
CFLAGS +=
CXXFLAGS +=
ASFLAGS +=
LDFLAGS += --specs=nano.specs --specs=nosys.specs
COMMONFLAGS += -mcpu=cortex-m4 -mthumb -mfpu=fpv4-sp-d16 -mfloat-abi=hard
LINKER_SCRIPT := $(BSP_ROOT)/STM32F4xxxx/LinkerScripts/STM32F429IG_flash.lds
```

2.8.2 更改链接文件

注销 MBED_APP_START 和 MBED_APP_SIZE 宏定义,并把宏定义用数值代替。

```
 7  HEAP_SIZE = 0x6000;
 8
 9  /*#if !defined(MBED_APP_START)*/
10  /*  #define MBED_APP_START 0x08000000*/
11  /*#endif*/
12
13  /*#if !defined(MBED_APP_SIZE)*/
14  /*  #define MBED_APP_SIZE 2048k*/
15  /*#endif*/
16
17  /* Specify the memory areas */
18  MEMORY
19  {
20      VECTORS (rx) : ORIGIN = 0x08000000, LENGTH = 0x400
21      FLASH (rx)   : ORIGIN = 0x08000000 + 0x400, LENGTH = 0x200000 - 0x400
22      CCM (rwx)    : ORIGIN = 0x10000000, LENGTH = 64K
23      RAM (rwx)    : ORIGIN = 0x20000000, LENGTH = 192k
24  }
```

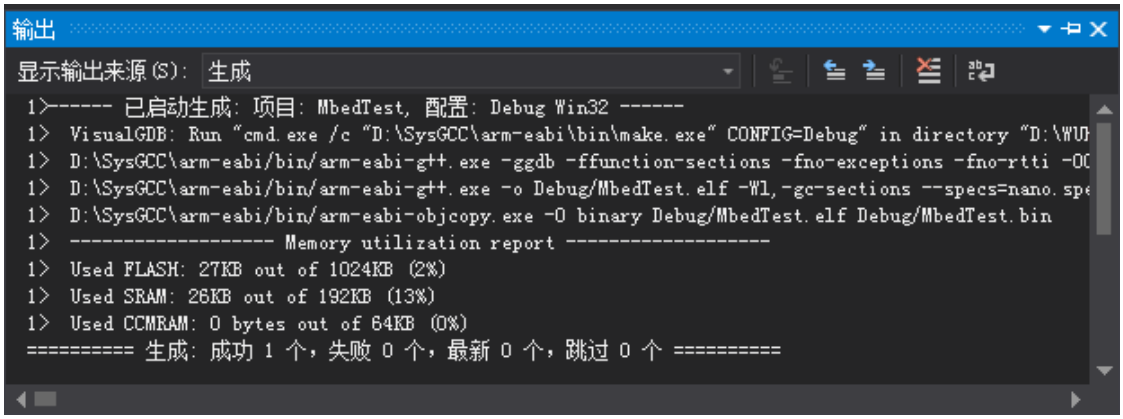
2.8.3 更改启动文件

放开 bl main 语句，以便程序启动后进入 main.cpp 文件中主函数。

```
/* Call the clock system intitialization function.*/
  bl SystemInitPre
  bl HAL_InitPre
  bl  SystemInit
/* Call static constructors */
 // bl __libc_init_array
/* Call the application's entry point.*/
 bl  main
  // Calling the crt0 'cold-start' entry point. There __libc_init_array is called
  // and when existing hardware_init_hook() and software_init_hook() before
  // starting main(). software_init_hook() is available and has to be called due
  // to initializsation when using rtos.
  bl _start
  bx  lr
.size  Reset_Handler, .-Reset_Handler
```

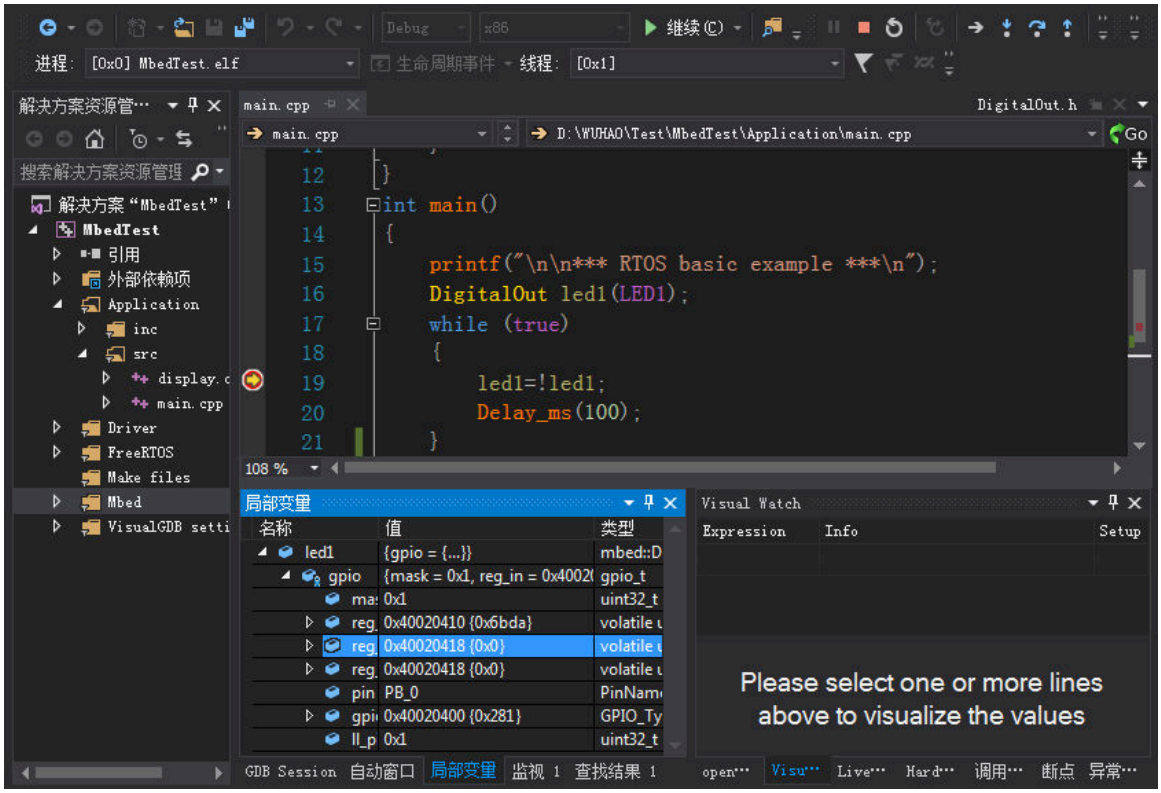
2.9 编译工程

一切设置正确后，点 VS2015 生成解决方案，经过慢长的等待后编译成功。显示 ROM 使用 27K，RAM 使用量 26K，对现在主流的芯片应用问题不大，可以接受。



2.10 调试工程

点“本地 Windows 调试器”，工程进入调试状态，OpenOCD 窗口输出调试信息，可设置断点或全速运行，全速运行后开发板 LED 灯开始闪烁，说明整个移植过程成功，并且能正常下载固件。



3 初识 Mbed 架构

3.1 通用 IO 口

在移植过程中，main 函数中只是定义灯 LED1，并没有对灯进行初始化，就可以执行闪灯动作。

DigitalOut led1(LED1);

DigitalOut 是一个驱动类，位于\Drivers\DigitalOut.h 文件，带参数 PinName 构造的构造函数，实际上是调用位于\Hal\mbed_gpio.c 中 gpio_init_out()，gpio_init()位于\targets\TARGET_STM\gpio_api.c，其中最

重要的是调用 Set_GPIO_Clock()根据 Pin 参数开启端口电源，并把操作 IO 口实际对象赋予 DigitalOut 类中的保护变量 gpio 并完成初始化。DigitalOut 类实现了读写和重载=号。所以 Led=! Led1 调用 read、write 和’=’重载，从而实现 LED 灯闪亮。

调用层次：Drivers->Hal->Target，目标层根据不同厂商芯片和芯片种类需要更改。通过三层调用实现与硬件无关。

重要变量类型：

PinName:枚举类型，芯片管脚定义，如 LED1 = PB_0

PinMode:枚举类型，管脚模式定义，上拉、下拉等

PinDirection：枚举类型，管脚输入输出方向定义

重要结构体：此结构体与硬件相关

```
typedef struct
{
    uint32_t mask;
    __IO uint32_t *reg_in;
    __IO uint32_t *reg_set;
    __IO uint32_t *reg_clr;
    PinName  pin;
    GPIO_TypeDef *gpio;
    uint32_t ll_pin;
} gpio_t;
```

```
class DigitalOut {

public:
    /** Create a DigitalOut connected to the specified pin
     *
     *  @param pin DigitalOut pin to connect to
     */
    DigitalOut(PinName pin) : gpio()
    {
        // No lock needed in the constructor
        gpio_init_out(&gpio, pin);
    }
```

```
void gpio_init_out(gpio_t *gpio, PinName pin)
{
    gpio_init_out_ex(gpio, pin, 0);
}

void gpio_init_out_ex(gpio_t *gpio, PinName pin, int value)
{
    _gpio_init_out(gpio, pin, PullNone, value);
}
```

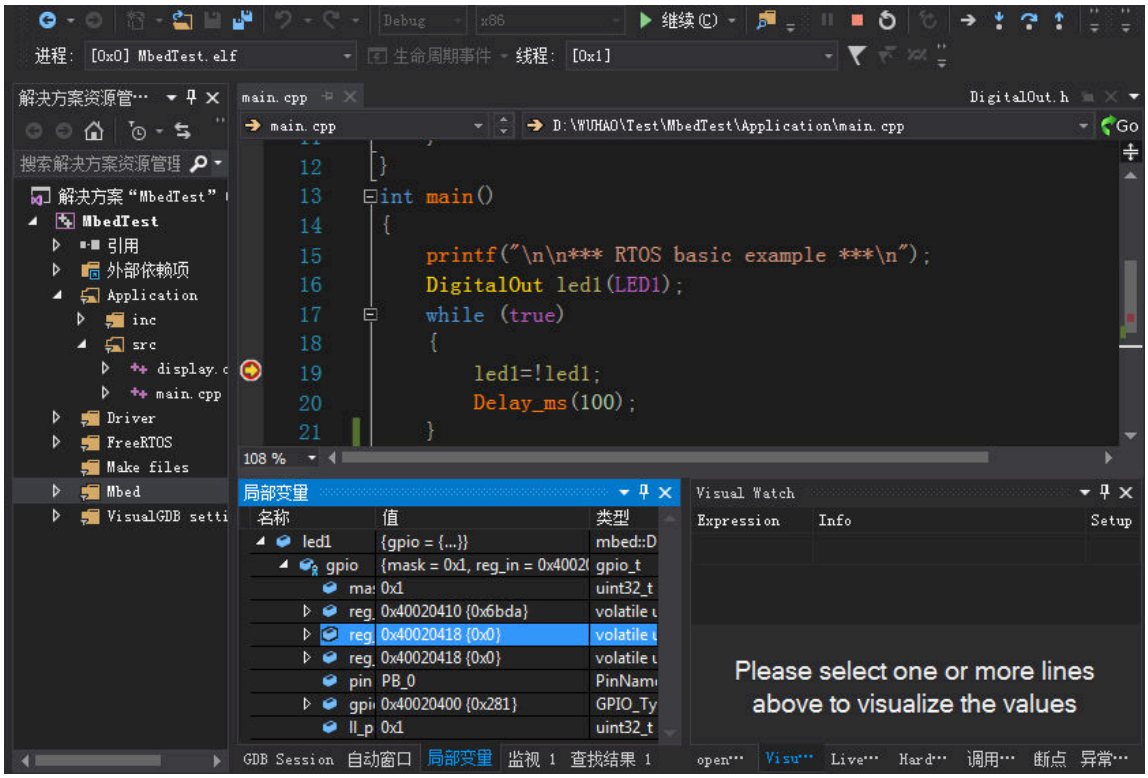
```
static inline void _gpio_init_out(gpio_t *gpio, PinName pin, PinMode mode, int value)
{
    gpio_init(gpio, pin);
    if (pin != NC) {
        gpio_write(gpio, value);
        gpio_dir(gpio, PIN_OUTPUT);
        gpio_mode(gpio, mode);
    }
}
```

```
void gpio_init(gpio_t *obj, PinName pin)
{
    obj->pin = pin;
    if (pin == (PinName)NC) {
        return;
    }

    uint32_t port_index = STM_PORT(pin);

    // Enable GPIO clock
    GPIO_TypeDef *gpio = Set_GPIO_Clock(port_index);

    // Fill GPIO object structure for future use
    obj->mask    = gpio_set(pin);
    obj->gpio    = gpio;
    obj->ll_pin  = ll_pin_defines[STM_PIN(obj->pin)];
    obj->reg_in  = &gpio->IDR;
    obj->reg_set = &gpio->BSRR;
#ifdef GPIO_IP_WITHOUT_BRR
    obj->reg_clr = &gpio->BSRR;
#else
    obj->reg_clr = &gpio->BRR;
#endif
}
```

3.2 调试信息输出

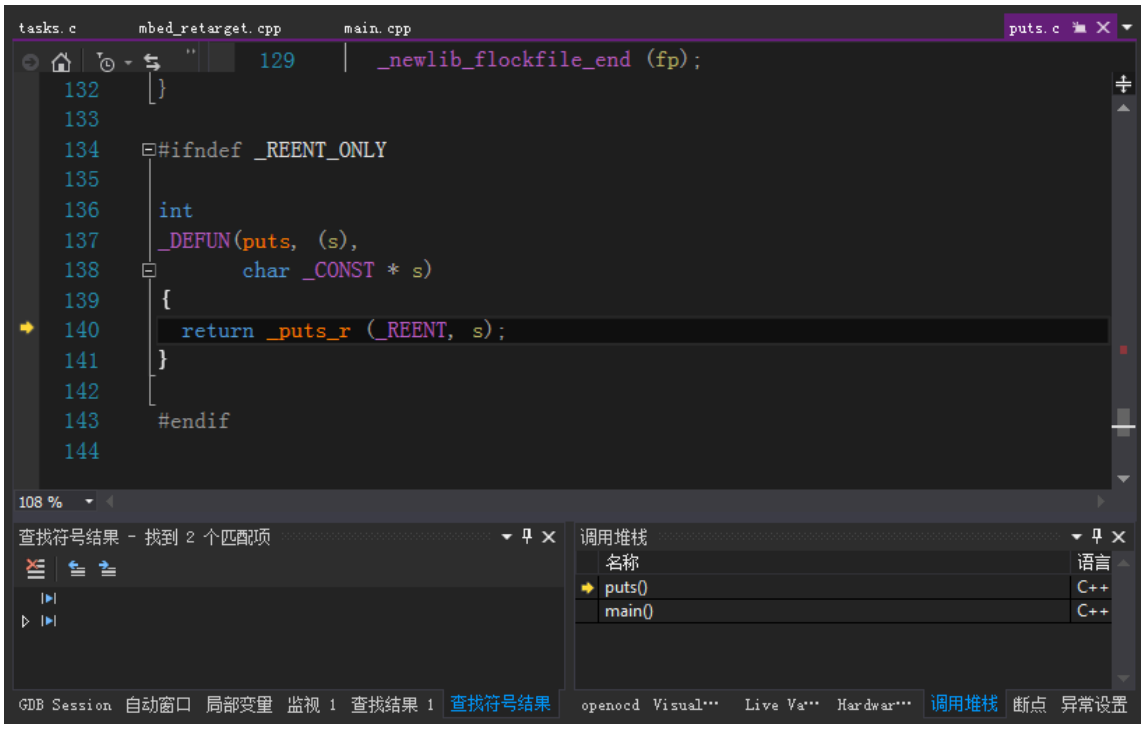
3.2.1 串口调试信息

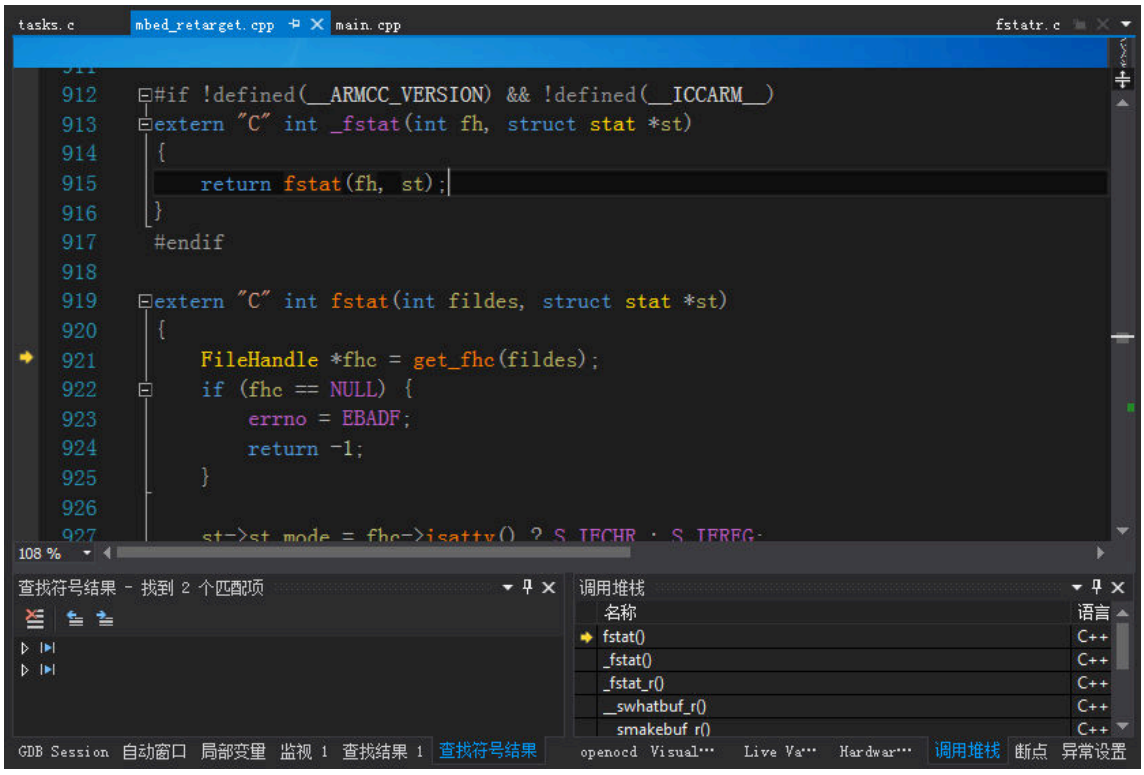
在移植过程调试语句 `printf("\n\n*** RTOS basic example ***\n");` 没有详细分析，当使用串口调试工具时，可以看到输出的调试信息。

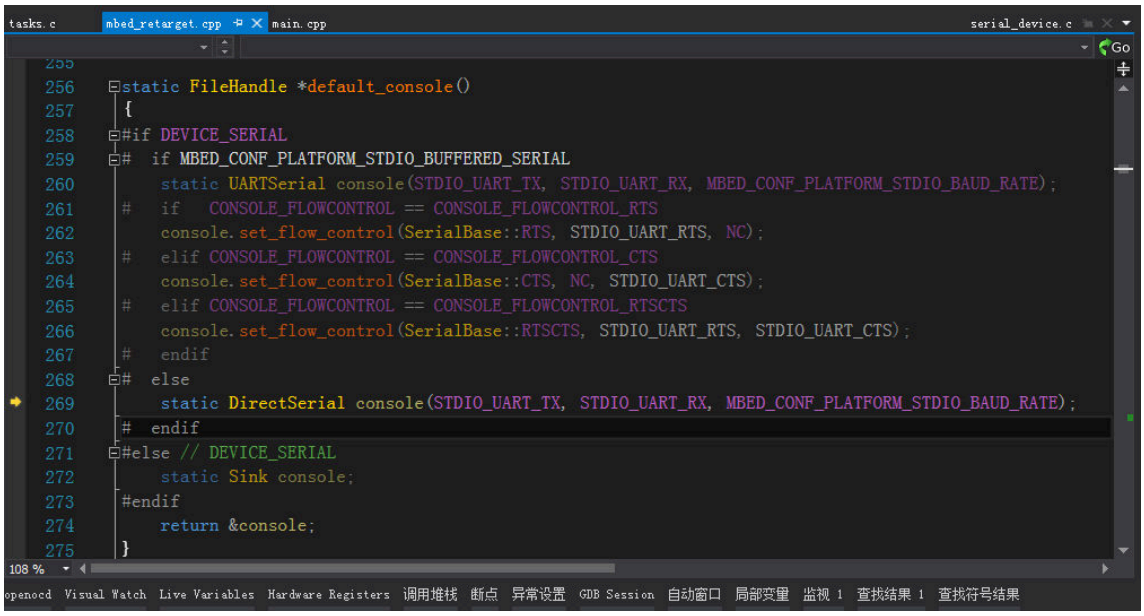


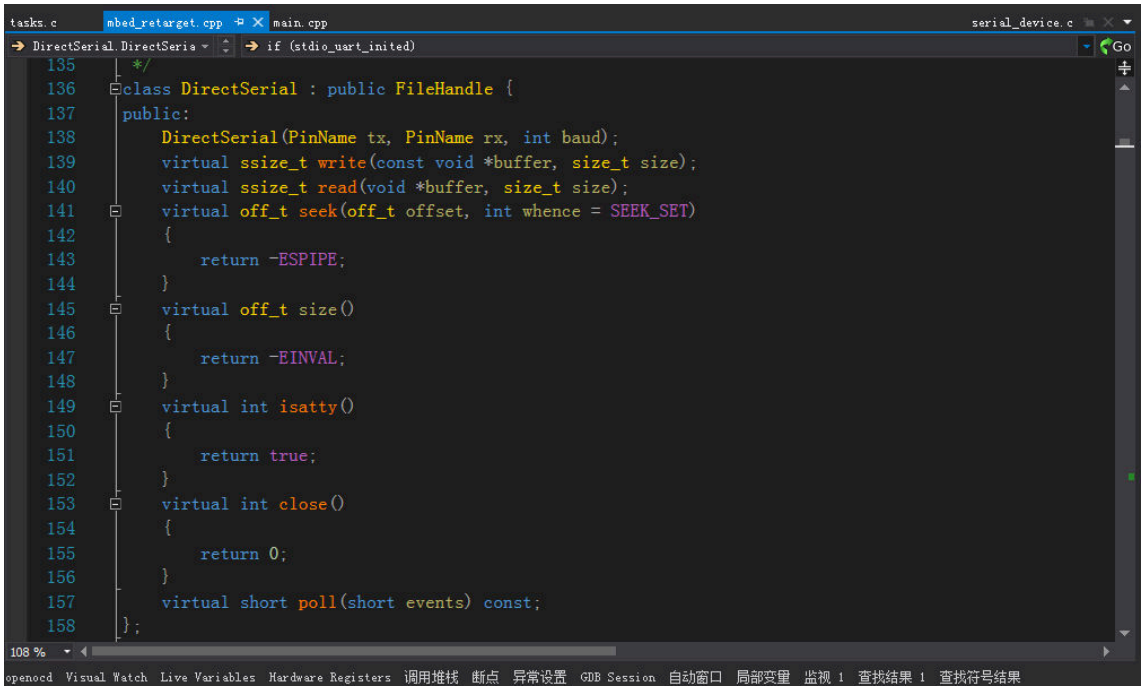
3.2.2 调试信息代码分析

经过一系列的调用系统库文件：puts-> _puts_r-> cantwrite-> __smakebuf_r-> _fstat_r-> _fstat-> fstat。


tasks.c
mbed_retarget.cpp
main.cpp
puts.c
129 _newlib_flockfile_end (fp);
132 }
133
134 #ifndef _REENT_ONLY
135
136 int
137 _DEFUN(puts, (s),
138 char _CONST * s)
139 {
140 return _puts_r (_REENT, s);
141 }
142
143 #endif
144
108 %
查找符号结果 - 找到 2 个匹配项
调用堆栈
名称
语言
puts()
C++
main()
C++
GDB Session
自动窗口
局部变量
监视 1
查找结果 1
查找符号结果
openocd
Visual···
Live Va···
Hardwar···
调用堆栈
断点
异常设置


tasks.c
mbed_retarget.cpp
main.cpp
fstatr.c
912 #if !defined(__ARMCC_VERSION) && !defined(__ICCARM__)
913 extern "C" int _fstat(int fh, struct stat *st)
914 {
915 return fstat(fh, st);
916 }
917 #endif
918
919 extern "C" int fstat(int fildes, struct stat *st)
920 {
921 FileHandle *fhc = get_fhc(fildes);
922 if (fhc == NULL) {
923 errno = EBADF;
924 return -1;
925 }
926
927 st->st_mode = fhc->isatty() ? S_IFCHR : S_IFREG;
108 %
查找符号结果 - 找到 2 个匹配项
调用堆栈
名称
语言
fstat()
C++
_fstat()
C++
_fstat_r()
C++
__swhatbuf_r()
C++
smakebuf_r()
C++
GDB Session
自动窗口
局部变量
监视 1
查找结果 1
查找符号结果
openocd
Visual···
Live Va···
Hardwar···
调用堆栈
断点
异常设置





3.3 外部中断

3.3.1 增加软件初始化

在引导文件中进入主函数之前增加 Software_init_hook，此函数主要完成把中断向量表拷贝到 RAM 中，完成系统时钟和 RTOS 相关初始化。

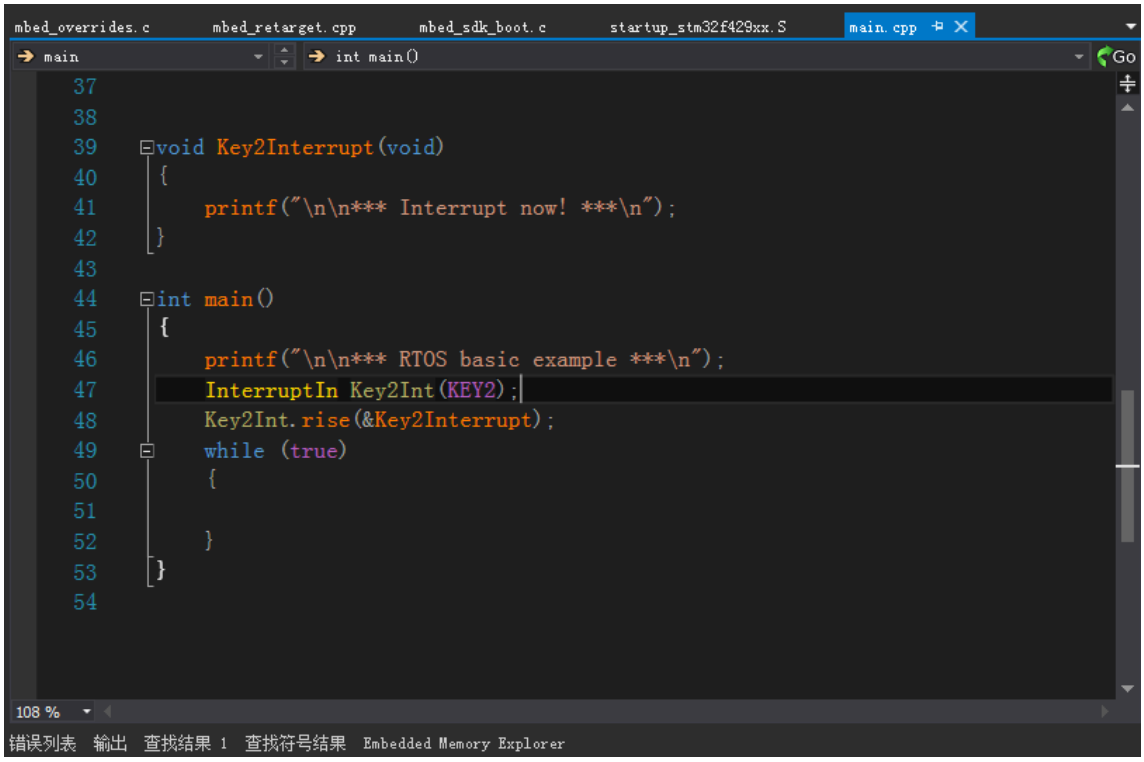
```
startup_stm32f429xx.S  main.cpp
103  LoopFillZerobss:
104    ldr  r3, = _ebss
105    cmp  r2, r3
106    bcc  FillZerobss
107
108  /* Call the clock system intitialization function.*/
109    bl SystemInitPre
110    bl HAL_InitPre
111    // SystemInit
112  /* Call static constructors */
113    bl __libc_init_array
114  /* Call the application's entry point.*/
115   bl software_init_hook
116   bl  main
117    // Calling the crt0 'cold-start' entry point. There __libc_init_array is called
118    // and when existing hardware_init_hook() and software_init_hook() before
119    // starting main(). software_init_hook() is available and has to be called due
120    // to initializsation when using rtos.
121    bl _start
122    bx  lr
123  .size  Reset_Handler, .-Reset_Handler
124
125  /**
126   * @brief  This is the code that gets called when the processor receives an
127   *         unexpected interrupt.  This simply enters an infinite loop, preserving
128   *         the system state for examination by a debugger.
```

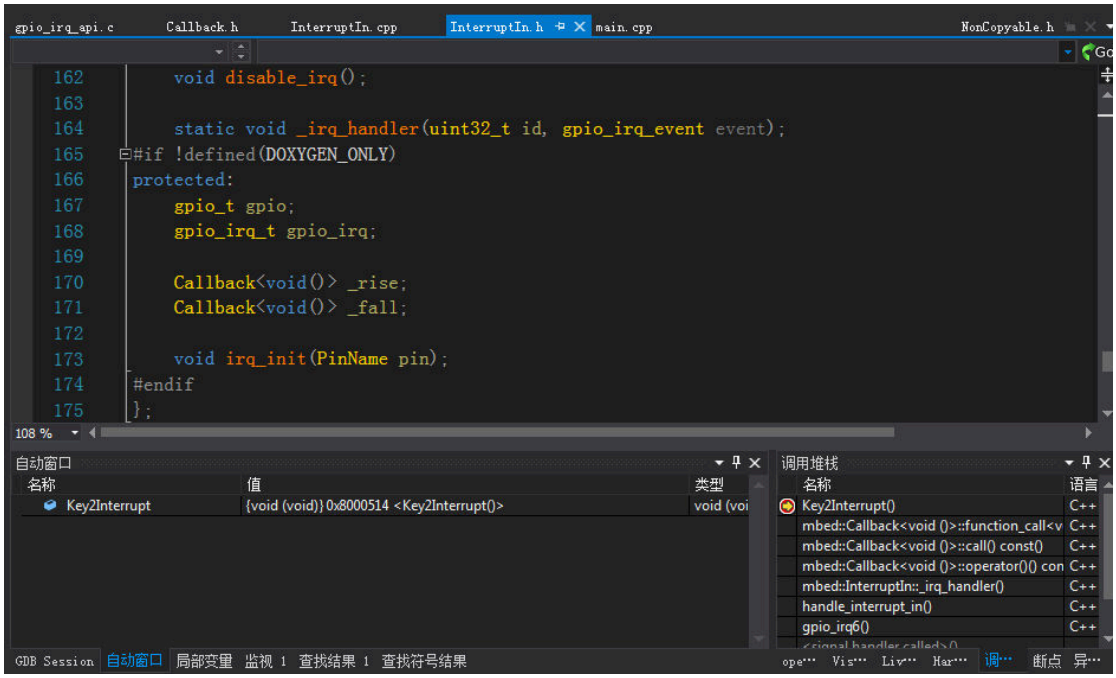
```
mbed_overrides.c  mbed_retarget.cpp  mbed_sdk_boot.c  startup_stm32f429xx.S  main.cpp
mbed_sdk_boot.c   D:\WUHAO\Test\MbedTest\Middlewares\ARMmbed\platform\mbed_sdk_boot.c
74  int $Sub$$main(void)
75  {
76      mbed_main();
77      return $Super$$main();
78  }
79
80  void _platform_post_stackheap_init(void)
81  {
82      mbed_copy_nvic();
83      mbed_sdk_init();
84  }
85
86  #elif defined (__GNUC__)
87
88  extern int __real_main(void);
89
90  void software_init_hook(void)
91  {
92      mbed_copy_nvic();
93      mbed_sdk_init();
94      software_init_hook_rtos();
95  }
96
```

3.3.2 定义外部中断

外部中断是由 InterruptIn 类实现和管理的，类中定义了上升沿（_rise）和下降沿(_fall)中断函数和中断服务总入口(_irq_handler)，KEY2 对应的 GPIO 为 PC13 管脚。

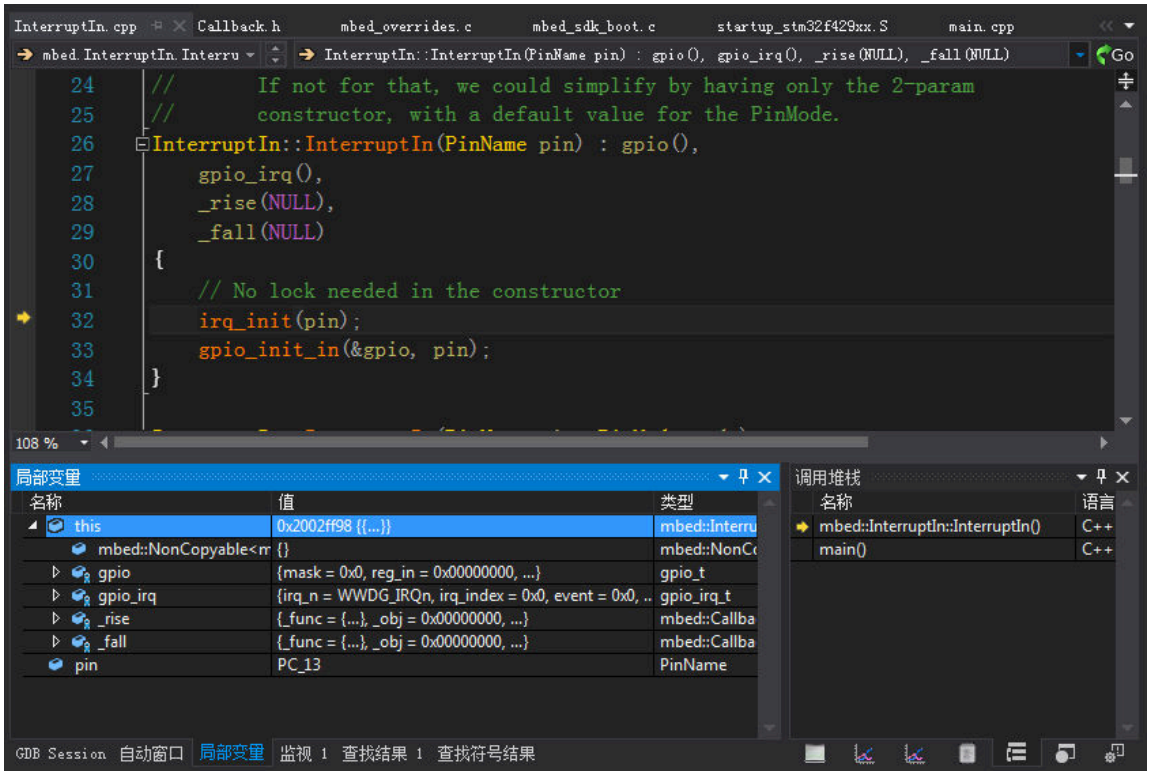
```
InterruptIn Key2Int(KEY2);      //定义外部中断引脚
Key2Int.rise(&Key2Interrupt); //上升沿中断，服务函数 Key2Interrupt
```

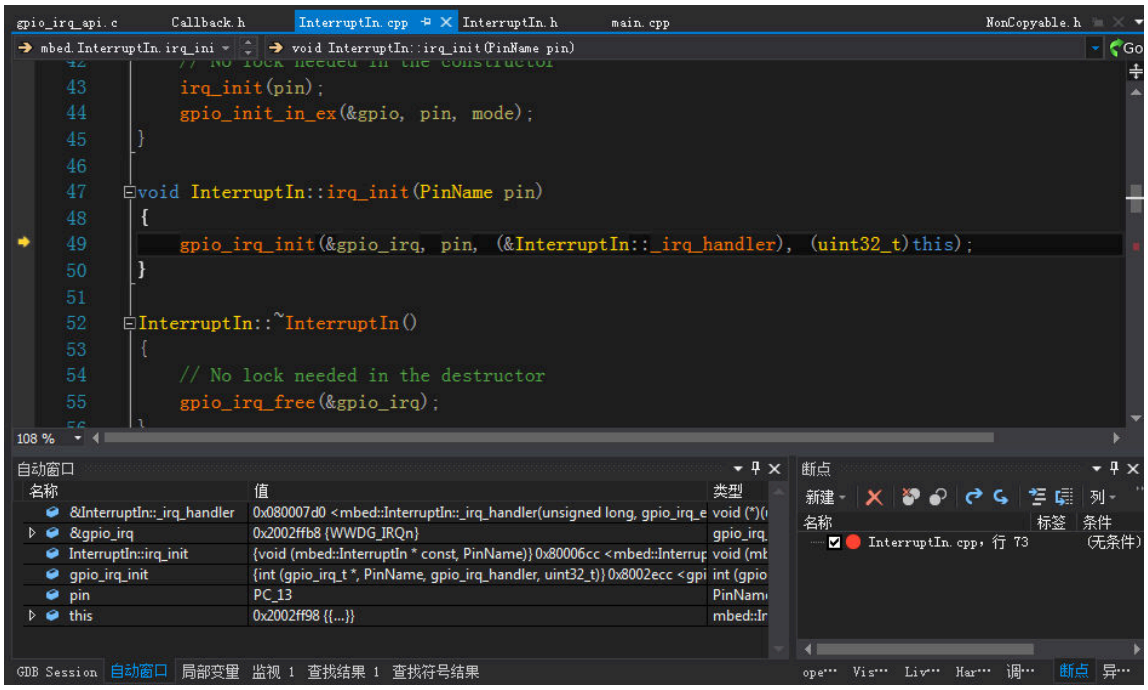


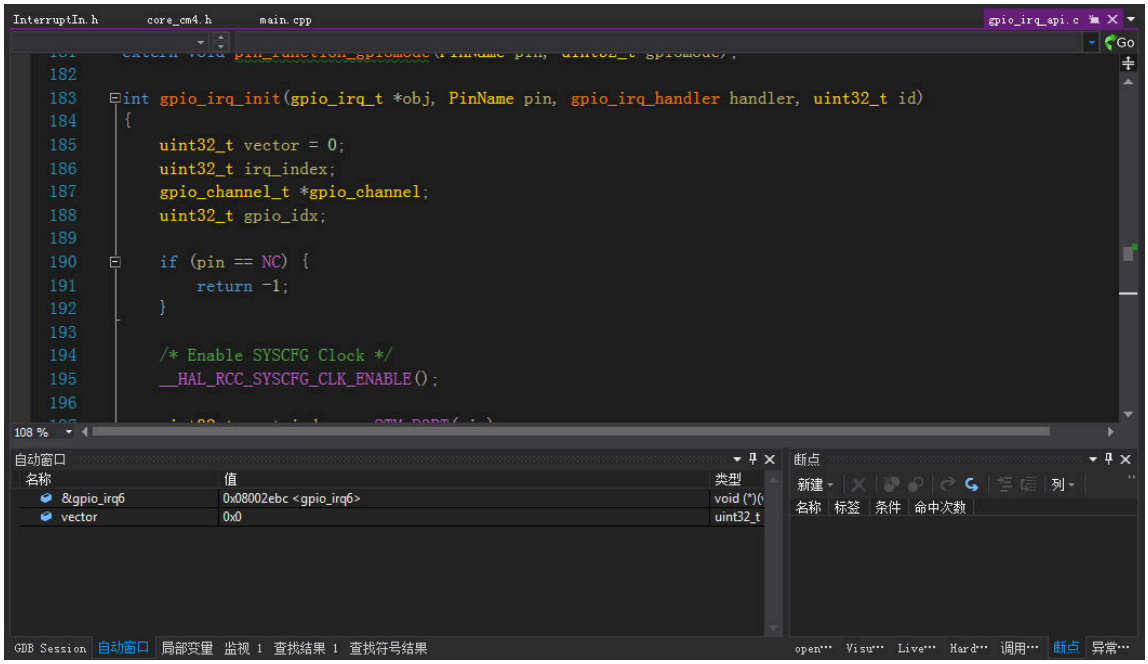


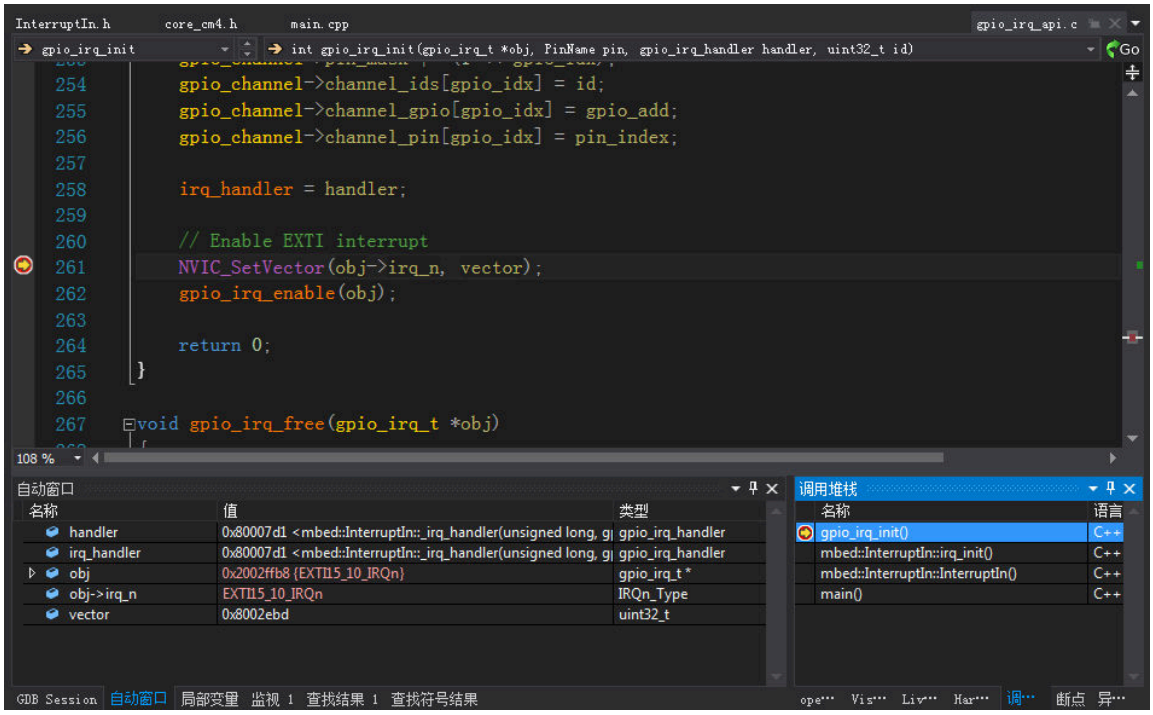
3.3.3 管脚和中断初始化

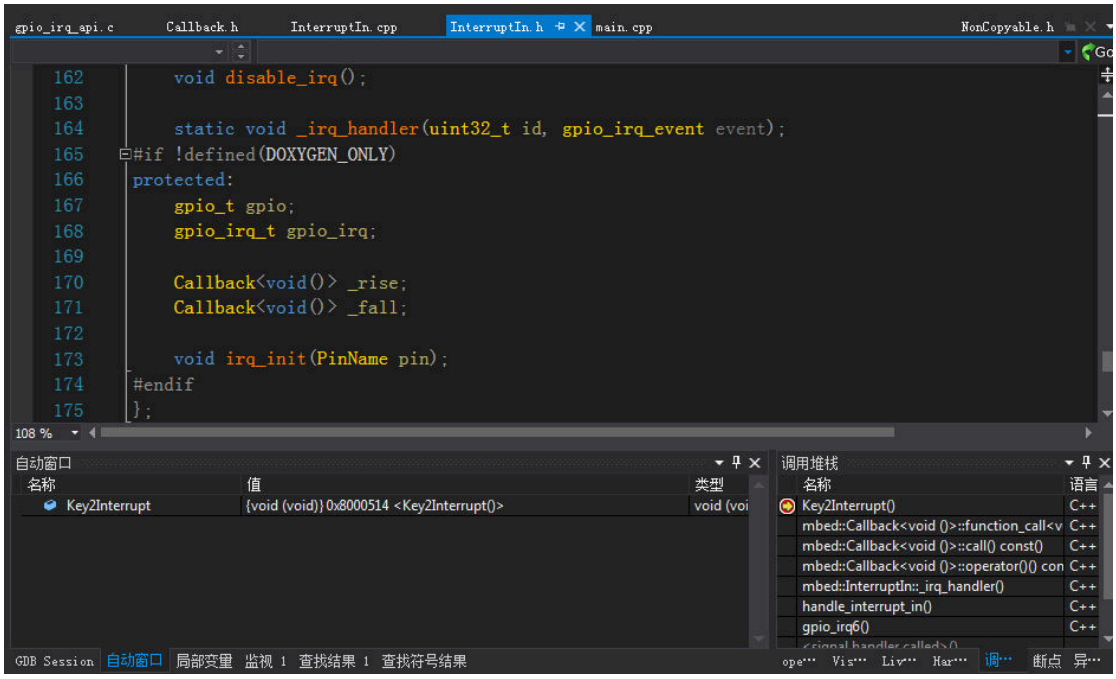
Irq_init(pin)完成调用硬件相关函数完成中断初始，并使用 InterruptIn 实例中断服务程序_irq_handler 初始化，根据 STM 手册 PC_13 中断号是 EXTI15_10_IRQn，在 Mbed 中断服务程序为 gpio_irq6，调用 NVIC_SetVector 把中断向量表更改为 gpio_irq6 入口并开启中断。gpio_init_in(&gpio，pin)初始管脚初始化。


InterruptIn.cpp
Callback.h
mbed_overrides.c
mbed_sdk_boot.c
startup_stm32f429xx.S
main.cpp
mbed.InterruptIn.Interru
InterruptIn::InterruptIn(PinName pin) : gpio(), gpio_irq(), _rise(NULL), _fall(NULL)
Go
24 // If not for that, we could simplify by having only the 2-param
25 // constructor, with a default value for the PinMode.
26 InterruptIn::InterruptIn(PinName pin) : gpio(),
27 gpio_irq(),
28 _rise(NULL),
29 _fall(NULL)
30 {
31 // No lock needed in the constructor
32 irq_init(pin);
33 gpio_init_in(&gpio, pin);
34 }
35
108 %
局部变量
名称
值
类型
this
0x2002ff98 {{...}}
mbed::Interru
mbed::NonCopyable<m
{}
mbed::NonC
gpio
{mask = 0x0, reg_in = 0x00000000, ...}
gpio_t
gpio_irq
{irq_n = WWDG_IRQn, irq_index = 0x0, event = 0x0, ..
gpio_irq_t
_rise
{_func = {...}, _obj = 0x00000000, ...}
mbed::Callba
_fall
{_func = {...}, _obj = 0x00000000, ...}
mbed::Callba
pin
PC_13
PinName
调用堆栈
名称
语言
mbed::InterruptIn::InterruptIn()
C++
main()
C++
GDB Session
自动窗口
局部变量
监视 1
查找结果 1
查找符号结果


gpio_irq_api.c
Callback.h
InterruptIn.cpp
InterruptIn.h
main.cpp
NonCopyable.h
mbed.InterruptIn.irq_ini
void InterruptIn::irq_init(PinName pin)
Go
43 irq_init(pin);
44 gpio_init_in_ex(&gpio, pin, mode);
45 }
46
47 void InterruptIn::irq_init(PinName pin)
48 {
49 gpio_irq_init(&gpio_irq, pin, (&InterruptIn::_irq_handler), (uint32_t)this);
50 }
51
52 InterruptIn::~InterruptIn()
53 {
54 // No lock needed in the destructor
55 gpio_irq_free(&gpio_irq);
108 %
自动窗口
名称
值
类型
&InterruptIn::_irq_handler
0x080007d0 <mbed::InterruptIn::_irq_handler(unsigned long, gpio_irq_e
void (*)(
&gpio_irq
0x2002ffb8 {WWDG_IRQn}
gpio_irq_
InterruptIn::irq_init
{void (mbed::InterruptIn * const, PinName)} 0x80006cc <mbed::Interrup
void (mb
gpio_irq_init
{int (gpio_irq_t *, PinName, gpio_irq_handler, uint32_t)} 0x8002ecc <gpi
int (gpio
pin
PC_13
PinNam
this
0x2002ff98 {{...}}
mbed::Ir
断点
新建
列
名称
标签
条件
InterruptIn.cpp，行 73
(无条件)
GDB Session
自动窗口
局部变量
监视 1
查找结果 1
查找符号结果
断点


InterruptIn.h
core_cm4.h
main.cpp
gpio_irq_api.c
182
183 int gpio_irq_init(gpio_irq_t *obj, PinName pin, gpio_irq_handler handler, uint32_t id)
184 {
185 uint32_t vector = 0;
186 uint32_t irq_index;
187 gpio_channel_t *gpio_channel;
188 uint32_t gpio_idx;
189
190 if (pin == NC) {
191 return -1;
192 }
193
194 /* Enable SYSCFG Clock */
195 __HAL_RCC_SYSCFG_CLK_ENABLE();
196
108 %
自动窗口
名称 值 类型
&gpio_irq6 0x08002ebc <gpio_irq6> void (*)(
vector 0x0 uint32_t
断点
新建
名称 标签 条件 命中次数
GDB Session 自动窗口 局部变量 监视 1 查找结果 1 查找符号结果
断点


InterruptIn.h
core_cm4.h
main.cpp
gpio_irq_api.c
gpio_irq_init
int gpio_irq_init(gpio_irq_t *obj, PinName pin, gpio_irq_handler handler, uint32_t id)
254 gpio_channel->channel_ids[gpio_idx] = id;
255 gpio_channel->channel_gpio[gpio_idx] = gpio_add;
256 gpio_channel->channel_pin[gpio_idx] = pin_index;
257
258 irq_handler = handler;
259
260 // Enable EXTI interrupt
261 NVIC_SetVector(obj->irq_n, vector);
262 gpio_irq_enable(obj);
263
264 return 0;
265 }
266
267 void gpio_irq_free(gpio_irq_t *obj)
108 %
自动窗口
名称 值 类型
handler 0x80007d1 <mbed::InterruptIn::_irq_handler(unsigned long, g gpio_irq_handler
irq_handler 0x80007d1 <mbed::InterruptIn::_irq_handler(unsigned long, g gpio_irq_handler
obj 0x2002ffb8 {EXTI15_10_IRQn} gpio_irq_t *
obj->irq_n EXTI15_10_IRQn IRQn_Type
vector 0x8002ebd uint32_t
调用堆栈
名称 语言
gpio_irq_init() C++
mbed::InterruptIn::irq_init() C++
mbed::InterruptIn::InterruptIn() C++
main() C++
GDB Session 自动窗口 局部变量 监视 1 查找结果 1 查找符号结果



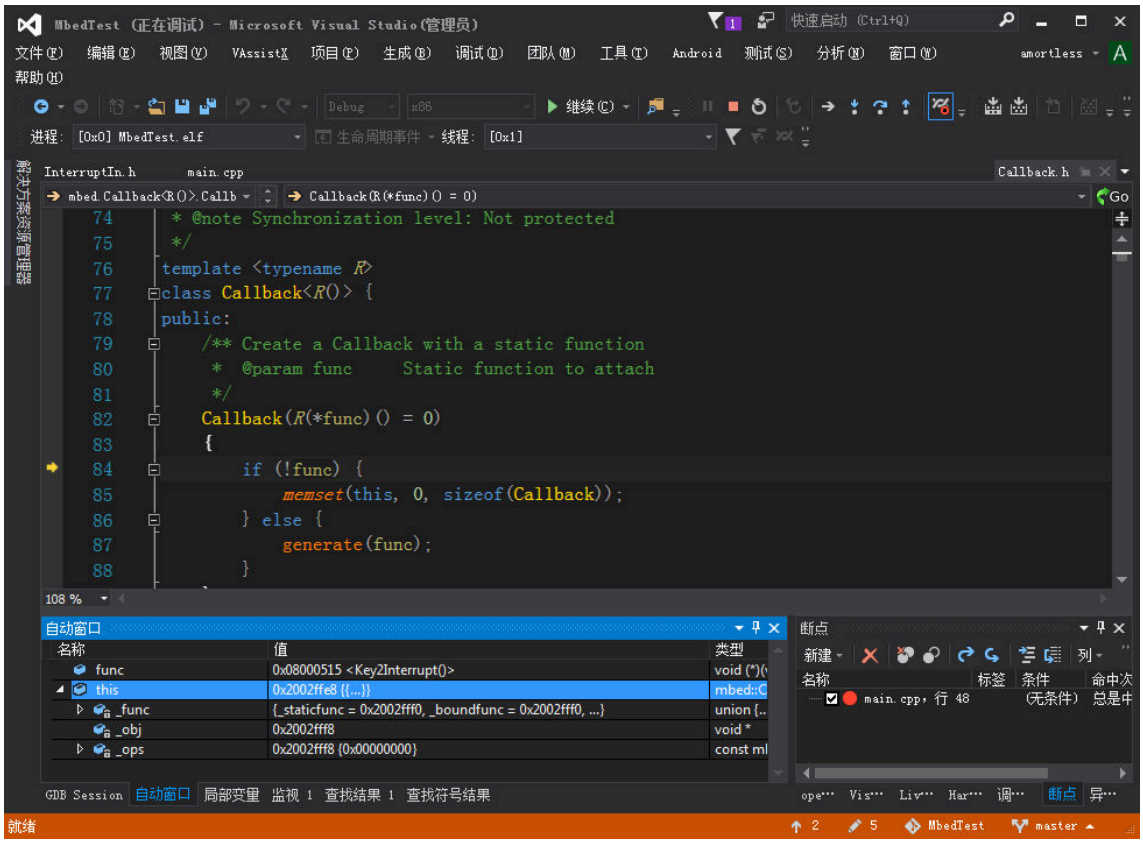
3.3.4 中断服务程序初始化

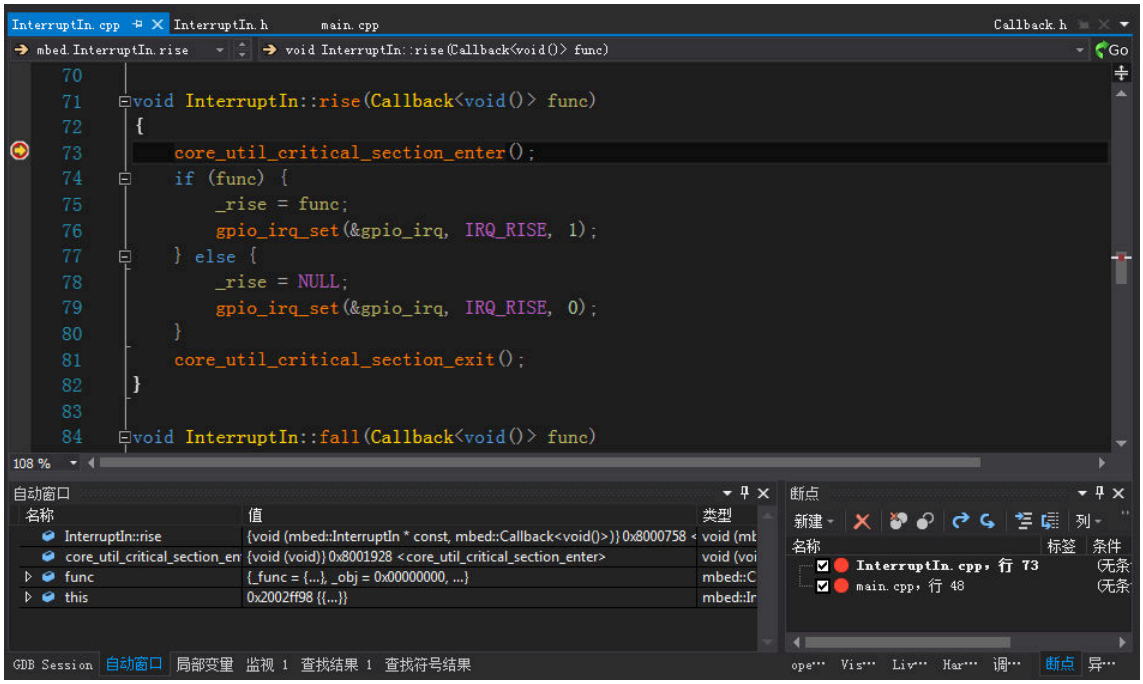
在 InterruptIn 类中下定义上升沿中断服务函数：

void InterruptIn::rise(Callback<void()> func),Callback 是中断服务模板类，把中断服务函数赋值到_rise。

template <typename R> class Callback<R()>，其中最重要的方法以：

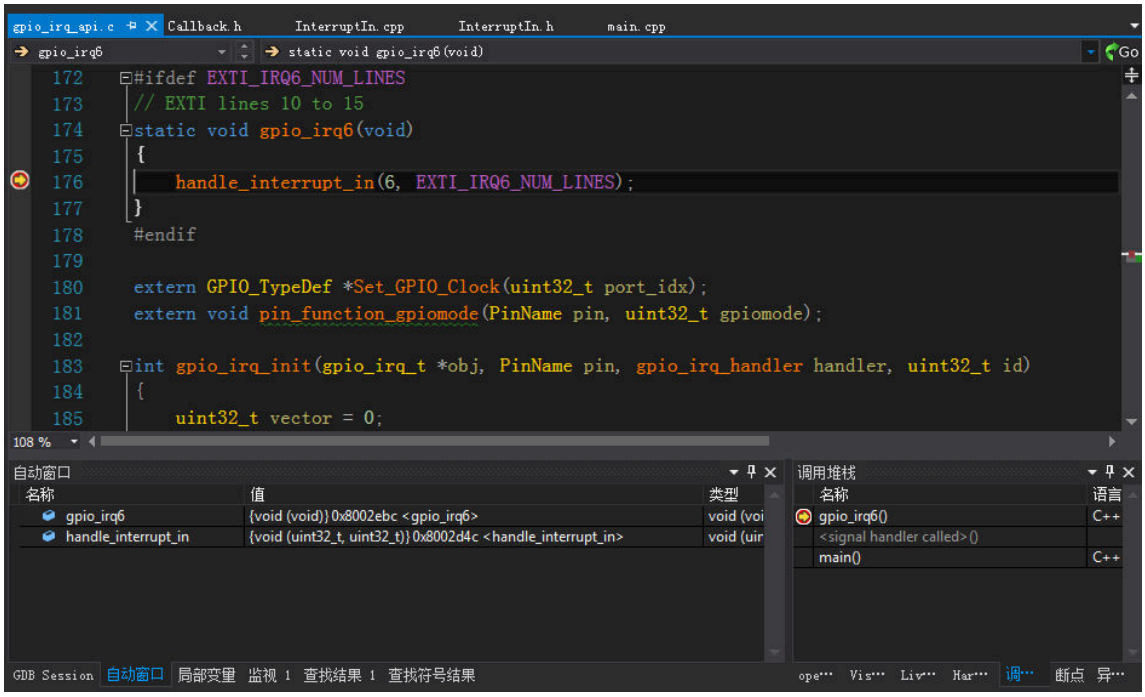
```
void attach(R(*func)());                    //添加绑定
R operator()() const;                       //调用真正的中断服务程序
static R function_call(const void *p)       //转跳到中断服务程序
```

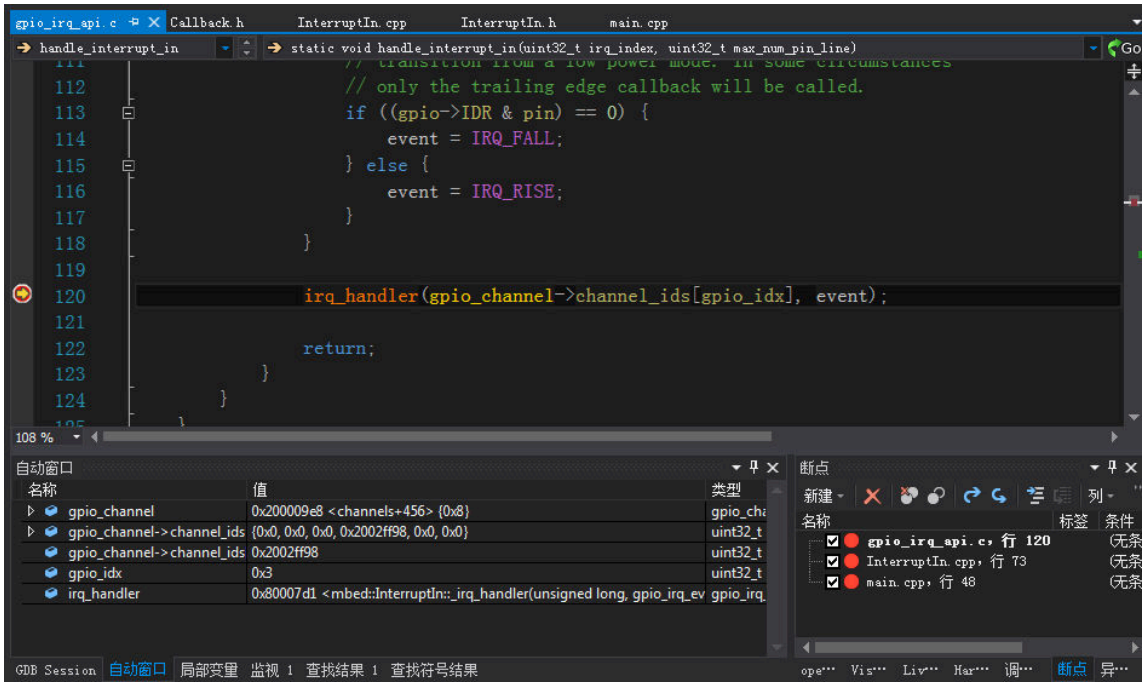


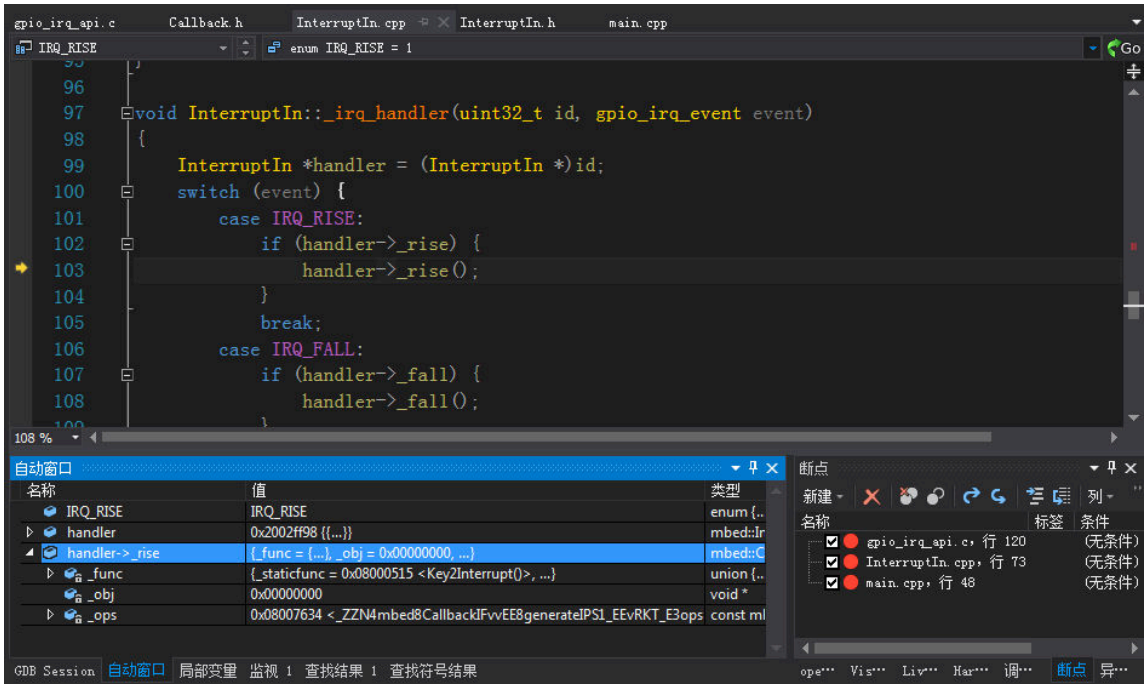


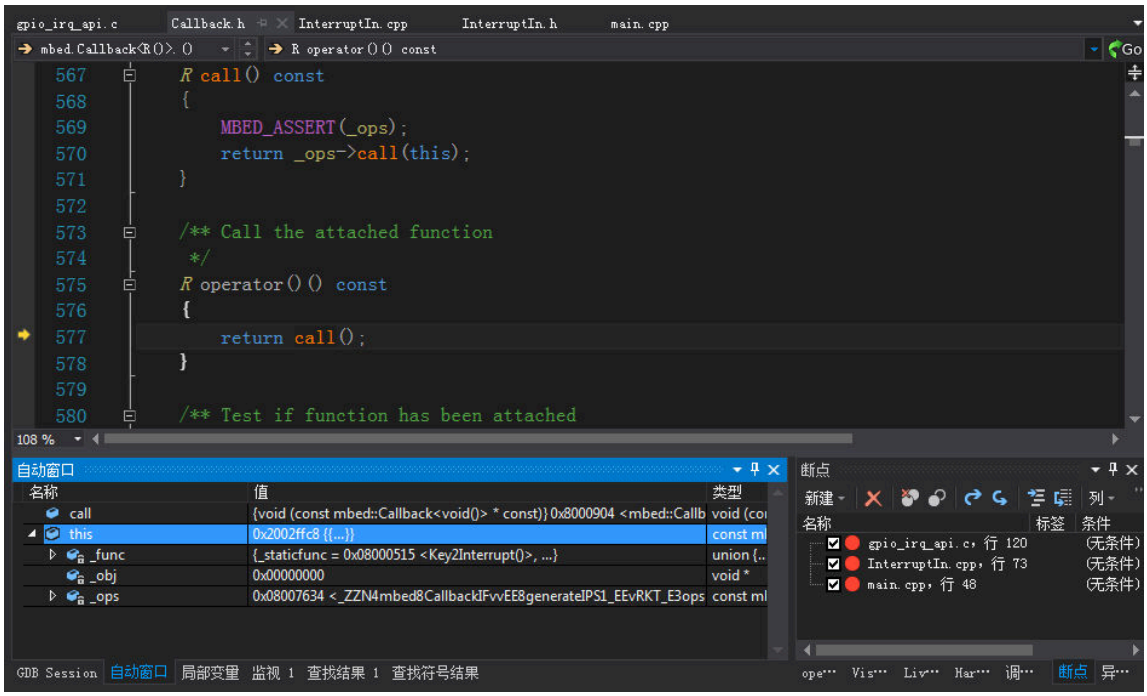
3.3.5 中断响应

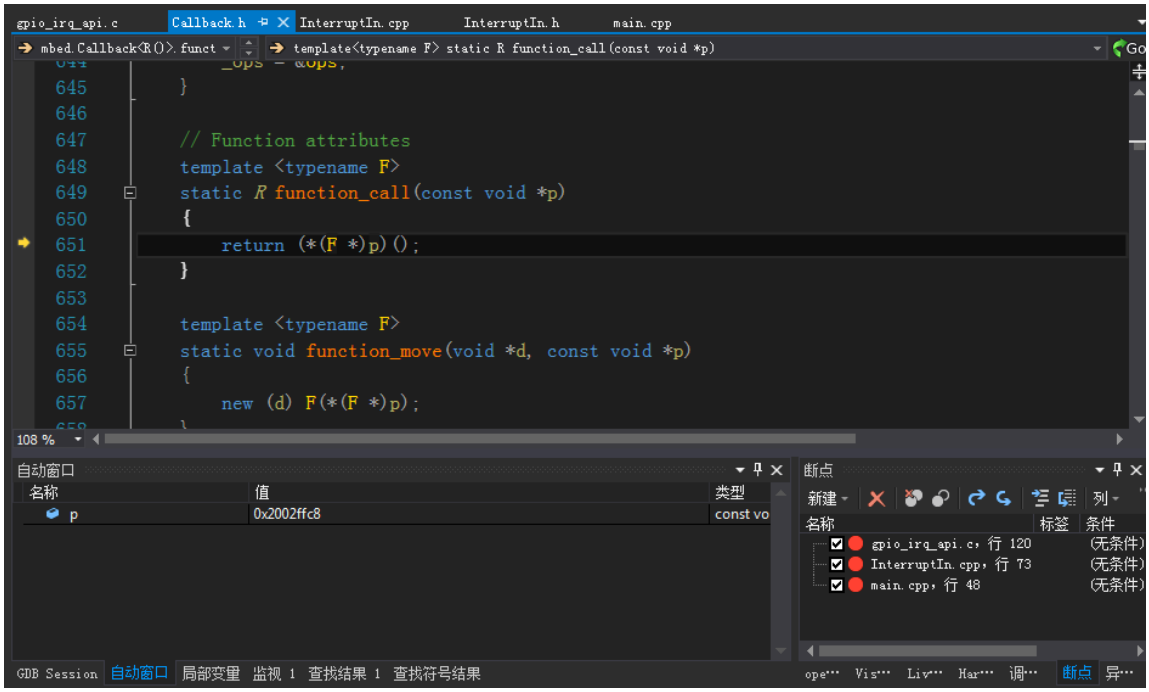
中断首先进入 gpio_irq6，调用 handle_interrupt_in，再调用 InterruptIn 类_irq_handler，根据中断初始化为上升沿调用_rise()，最后调用 Callback 类中 function_call 转跳到用户自定义中断服务函 Key2Interrupt()实现整个中断响应。

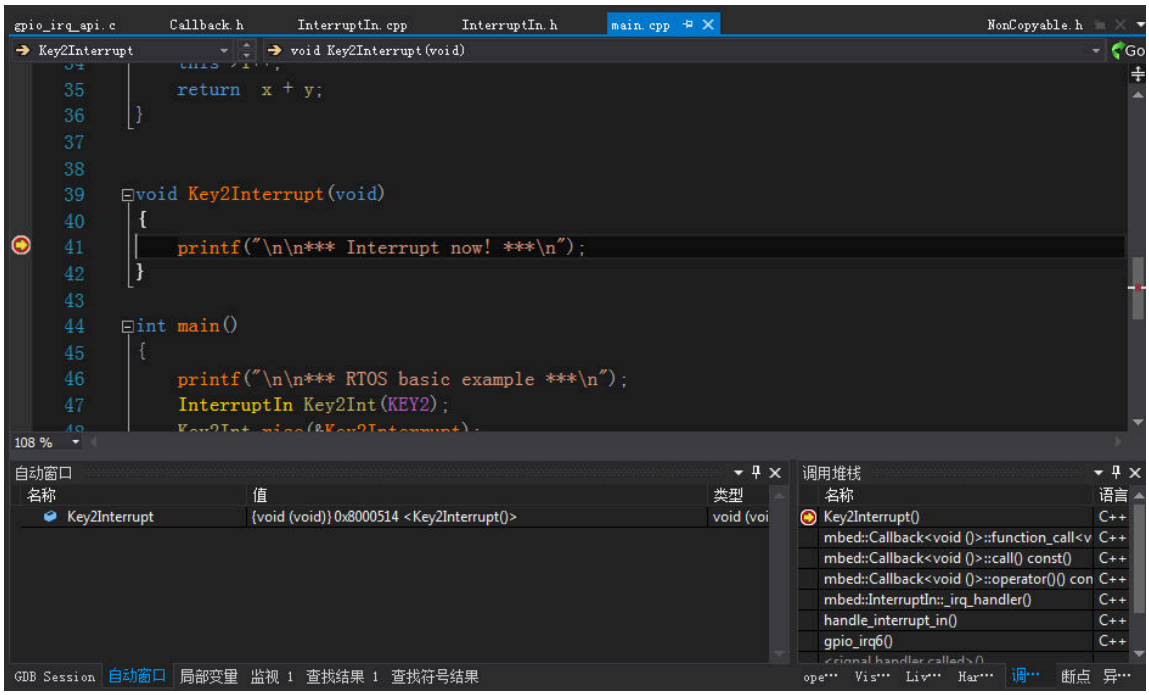

gpio_irq_api.c
Callback.h
InterruptIn.cpp
InterruptIn.h
main.cpp
gpio_irq6
static void gpio_irq6(void)
172 #ifdef EXTI_IRQ6_NUM_LINES
173 // EXTI lines 10 to 15
174 static void gpio_irq6(void)
175 {
176 handle_interrupt_in(6, EXTI_IRQ6_NUM_LINES);
177 }
178 #endif
179
180 extern GPIO_TypeDef *Set_GPIO_Clock(uint32_t port_idx);
181 extern void pin_function_gpiomode(PinName pin, uint32_t gpiomode);
182
183 int gpio_irq_init(gpio_irq_t *obj, PinName pin, gpio_irq_handler handler, uint32_t id)
184 {
185 uint32_t vector = 0;
108 %
自动窗口
名称
值
类型
gpio_irq6
{void (void)} 0x8002ebc <gpio_irq6>
void (voi
handle_interrupt_in
{void (uint32_t, uint32_t)} 0x8002d4c <handle_interrupt_in>
void (uir
调用堆栈
名称
语言
gpio_irq6()
C++
<signal handler called>()
main()
C++
GDB Session
自动窗口
局部变量
监视 1
查找结果 1
查找符号结果
断点
异…


gpio_irq_api.c
Callback.h
InterruptIn.cpp
InterruptIn.h
main.cpp
handle_interrupt_in
static void handle_interrupt_in(uint32_t irq_index, uint32_t max_num_pin_line)
// only the trailing edge callback will be called.
if ((gpio->IDR & pin) == 0) {
event = IRQ_FALL;
} else {
event = IRQ_RISE;
}
}
irq_handler(gpio_channel->channel_ids[gpio_idx], event);
return;
}
}
108 %
自动窗口
名称
值
类型
gpio_channel
0x200009e8 <channels+456> {0x8}
gpio_cha
gpio_channel->channel_ids
{0x0, 0x0, 0x0, 0x2002ff98, 0x0, 0x0}
uint32_t
gpio_channel->channel_ids
0x2002ff98
uint32_t
gpio_idx
0x3
uint32_t
irq_handler
0x80007d1 <mbed::InterruptIn::_irq_handler(unsigned long, gpio_irq_ev
gpio_irq_
断点
新建
列
名称
标签
条件
gpio_irq_api.c，行 120
InterruptIn.cpp，行 73
main.cpp，行 48
GDB Session
自动窗口
局部变量
监视 1
查找结果 1
查找符号结果
断点
异…


gpio_irq_api.c
Callback.h
InterruptIn.cpp
InterruptIn.h
main.cpp
IRQ_RISE
enum IRQ_RISE = 1
96
97 void InterruptIn::_irq_handler(uint32_t id, gpio_irq_event event)
98 {
99 InterruptIn *handler = (InterruptIn *)id;
100 switch (event) {
101 case IRQ_RISE:
102 if (handler->_rise) {
103 handler->_rise();
104 }
105 break;
106 case IRQ_FALL:
107 if (handler->_fall) {
108 handler->_fall();
108 %
自动窗口
名称
值
类型
IRQ_RISE
IRQ_RISE
enum {..
handler
0x2002ff98 {{...}}
mbed::Ir
handler->_rise
{_func = {...}, _obj = 0x00000000, ...}
mbed::C
_func
{_staticfunc = 0x08000515 <Key2Interrupt()>, ...}
union {..
_obj
0x00000000
void *
_ops
0x08007634 <_ZZN4mbed8CallbackIFvvEE8generateIPS1_EEvRKT_E3ops
const ml
GDB Session
自动窗口
局部变量
监视 1
查找结果 1
查找符号结果
断点
新建
列
名称
标签
条件
gpio_irq_api.c, 行 120
(无条件)
InterruptIn.cpp, 行 73
(无条件)
main.cpp, 行 48
(无条件)
断点


gpio_irq_api.c
Callback.h
InterruptIn.cpp
InterruptIn.h
main.cpp
mbed.Callback<R()>.()
R operator()() const
567 R call() const
568 {
569 MBED_ASSERT(_ops);
570 return _ops->call(this);
571 }
572
573 /** Call the attached function
574 */
575 R operator()() const
576 {
577 return call();
578 }
579
580 /** Test if function has been attached
108 %
自动窗口
名称
值
类型
call
{void (const mbed::Callback<void()> * const)} 0x8000904 <mbed::Callb
void (co
this
0x2002ffc8 {{...}}
const ml
_func
{_staticfunc = 0x08000515 <Key2Interrupt()>, ...}
union {..
_obj
0x00000000
void *
_ops
0x08007634 <_ZZN4mbed8CallbackIFvvEE8generateIPS1_EEvRKT_E3ops
const ml
GDB Session
自动窗口
局部变量
监视 1
查找结果 1
查找符号结果
断点
新建
列
名称
标签
条件
gpio_irq_api.c, 行 120
(无条件)
InterruptIn.cpp, 行 73
(无条件)
main.cpp, 行 48
(无条件)
断点

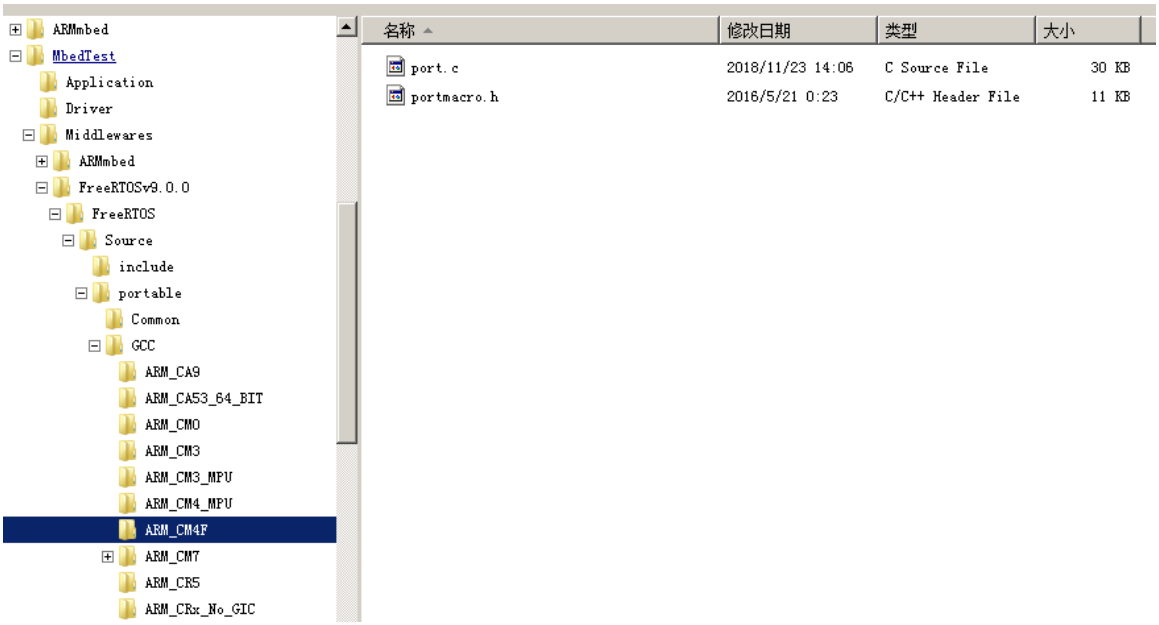


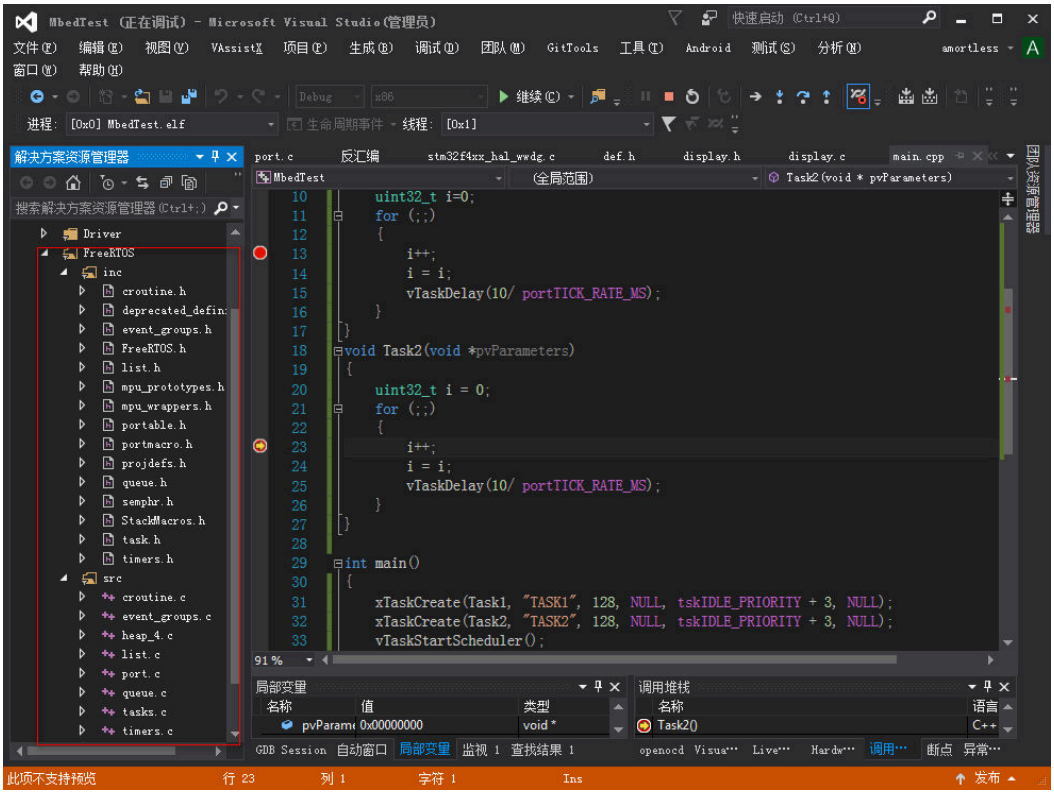


4 FreeRTOS 移植

4.1 增加 FreeRTOS 源代码

对于 STM32F429IG 控制器，port.c 位于 ARM_CM4F 目录，此文件与硬件有关，内存管理使用 heap_4.c，添加 FreeRTOS 所有源代码。



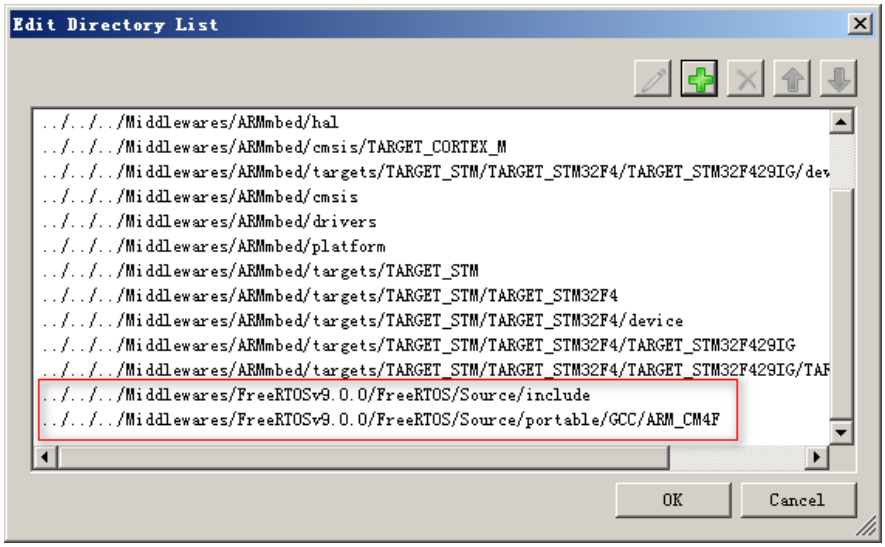


4.2 修改启动文件

修改启动文件 startup_stm32f429xx.S，修改 vPortSVCHandler、xPortPendSVHandler 和 xPortSysTickHandler。

```
g_pfnVectors:
  .word  __stack
  .word  Reset_Handler

  .word  NMI_Handler
  .word  HardFault_Handler
  .word  MemManage_Handler
  .word  BusFault_Handler
  .word  UsageFault_Handler
  .word  0
  .word  0
  .word  0
  .word  0
  .word  vPortSVCHandler
  .word  DebugMon_Handler
  .word  0
  .word  xPortPendSVHandler
  .word  xPortSysTickHandler
```

4.3 修改工程头文件目录



4.4 增加进程

在主文件 main.cpp 添加 Task1 和 Task2 两个任务并在主函数中创建，使用 vTaskStartScheduler 启动操作系统运行并进行任务切换。

4.5 调试工程

编译完成后，点“本地 Windows 调试器”，工程进入调试状态，并在 Task1 和 Task2 任务中设置断点，全速运行后调试点分别停在两个任务中，此时表示 FreeRTOS 操作系统移植成功。