# 第18章 嵌入式操作系统原理及应用

在一个真正的嵌入式系统中，必须要搭载操作系统。本章将通过在STM32G071 MCU上移植和运行国产操作系统RT-Thread来说明嵌入式操作系统的基本原理，以及它在嵌入式系统中的作用。

## 18.1 操作系统的必要性

从本书前面所介绍内容可知，在STM32G071 MCU上开发应用程序时常使用单任务程序和轮询程序。这些程序/应用都不带有操作系统（operating system，OS），就是我们经常所说的在MCU上所开发的“裸奔“程序。

虽然使用ST提供的HAL和LL提高了应用程序的开发效率，但是使用单任务程序或轮询程序仍然没有解决一些应用程序开发中的根本问题。

### 18.1.1 单任务程序

一个标准的C程序用main函数启动执行。在嵌入式应用中，main通常作为一个无限循环，将其看作一个单任务，这个任务连续运行。

【例18-1】 单任务的C语言描述，如代码清单18-1所示。

**代码清单18-1 单任务程序**

int counter;

void main (void) {

counter = 0;

while (1) { //无限循环

counter++; //递增计数器

}

}

很明显，一旦开始CPU运行这个程序，除非强行退出，永远无法释放CPU资源，其他程序永远不能得到CPU的服务（执行）。

### 18.1.2 轮询程序

不用实时操作系统（real-time operating system，RTOS），解决单任务程序的一个方法就是将需要CPU执行的一些程序编写成为子程序，然后用一个轮询预安排的多任务机制，实现一个更复杂的C程序。在这个机制中，在一个无限循环中重复地调用任务或者函数。

【例18-2】轮询的C语言描述，如代码清单18-2所示。

**代码清单18-2 轮询程序**

int counter;

void main (void) {

counter = 0;

while (1) { //无限循环

check\_serial\_io (); //处理串行I/O设备

process\_serial\_cmds (); //处理串行输入

check\_kbd\_io (); //检查键盘I/O设备

process\_kbd\_cmds (); //处理键盘输入

adjust\_ctrlr\_parms (); //调整控制器

counter++; //递增计数器

}

}

与前面的单任务程序相比，本质还是一样，即除非强行退出，CPU永远运行该程序。但是，改进之处就是，在运行程序的时候，可以按一定的顺序轮流的执行其他功能，如串行处理、键盘输入等。但是，响应事件的能力较差，如硬件上已经出现了键盘按键的事件，但是必须等待程序轮询执行到process\_kbd\_cmds子程序时才能进行处理。**延时**

从这两个例子可以看出，单任务和轮询的程序运行方式效率很低、响应事件的时间较长，并且不能同时运行多个程序，以及不能有效地管理计算机的硬件资源。因此，就需要引入操作系统来解决这些问题。

## 18.2 操作系统基本知识

[**操作系统**](http://baike.baidu.com/subview/880/4940471.htm)**是管理和控制**[**计算机**](http://baike.baidu.com/view/3314.htm)[**硬件**](http://baike.baidu.com/view/25278.htm)**与**[**软件**](http://baike.baidu.com/subview/37/6030295.htm)**资源的**[**计算机**](http://baike.baidu.com/view/3314.htm)**程序，是直接运行在计算机硬件上的最基本的**[**系统软件**](http://baike.baidu.com/view/7860.htm)**，任何其他软件都必须在**[**操作系统**](http://baike.baidu.com/subview/880/4940471.htm)**的支持下才能运行。**

操作系统运行在硬件系统上，它驻留在内存中，并给上层提供两种接口，即操作接口和编程接口。操作接口由一系列操作指令组成，用户通过操作接口可以方便地使用计算机。编程接口由一系列的系统调用组成各种程序可以使用这些系统调用让操作系统为其服务，并通过操作系统来使用硬件和软件资源。在计算机中通过操作系统提供的功能，就可以运行其他应用程序。

### 18.2.1 操作系统的作用

操作系统的作用主要体现在以下两方面：

（1）屏蔽硬件物理特性和操作细节，为用户使用计算机提供了便利。对于一个复杂的CPU来说，它的指令集中包含大量的机器指令，在CPU中通过微指令控制序列来实现这些机器指令。**早期**的计算机程序开发人员就是**在计算机硬件上**直接通过**汇编语言和C语言编写程序**。这种方式在早期的计算机系统中没有任何问题，但是随着计算机硬件体系结构越来越复杂，这种直接在计算机硬件上编程的设计方式就会遇到很多困难，比如如何高效地管理计算机硬件系统的功能部件（包括存储器、外设等）。

（2）有效管理系统资源以提高系统资源使用效率。如何有效地管理和合理地分配系统资源，提高系统资源的使用效率是操作系统必须发挥的主要作用。

**资源利用率**和**系统吞吐量**是衡量计算机性能的两个重要的指标满足这两个性能指标，这就要求为多个要运行的程序提供和分配计算机资源。当在一个计算机系统中运行多个程序的时候，就需要解决资源共享问题，以及如何分配、管理有限的资源。

### 18.2.2 操作系统的功能

操作系统位于底层硬件与用户之间，是两者沟通的桥梁。用户可以通过操作系统的用户界面输入指令。操作系统则对指令进行解释，驱动硬件设备，实现用户要求。

通常，一个完整的OS应该提供以下的功能。

#### 1．资源管理

根据用户需求，操作系统按一定的策略来分配和调度系统的设备资源和信息资源。操作系统中的存储管理模块就负责分配应用程序所需要的内存单元，以便可以在计算机上执行该应用程序，在程序执行结束后将它占用的内存单元收回以便进行重新分配。对于提供虚拟存储资源的计算机系统，操作系统还要与硬件配合做好页面调度工作，根据执行程序的要求分配页面，在执行中将页面调入和调出内存以及回收页面等。

处理器管理或称处理器调度，是操作系统资源管理功能的另一个重要内容。在一个允许多个程序同时执行的系统里，操作系统会根据一定的策略将处理器交替地分配给系统内等待运行的程序。一个等待运行的程序只有在获得了处理器后才能运行。一个程序在运行中若遇到某个事件，例如启动外部设备而暂时不能继续运行下去，或发生一个外部事件等，操作系统就要来处理相应的事件，然后重新分配处理器。

操作系统的设备管理功能主要是分配和回收外部设备以及控制外部设备按用户程序的要求进行操作等。对于非存储型外部设备，如[打印机](http://baike.baidu.com/view/7836.htm)、[显示器](http://baike.baidu.com/view/18610.htm)等，它们可以直接作为一个设备分配给一个用户程序，在使用完毕后回收以便给另一个需求的用户使用。对于存储型的外部设备，如磁盘、磁带等，则是给用户提供存储空间，以保存用户的文件和数据。存储性外部设备的管理与信息管理是紧密联系的。

信息管理是操作系统的一个重要的功能，主要是向用户提供一个[文件系统](http://baike.baidu.com/view/266589.htm)。一般说，一个文件系统向用户提供创建文件、撤销文件、读写文件、打开和关闭文件等功能。有了文件系统后，用户可按文件名存取数据而无需知道这些数据存放在哪里。这种做法不仅便于用户使用而且还有利于用户共享公共数据。此外，由于文件建立时允许创建者规定使用权限，这就可以保证数据的安全性。

#### 2．程序控制

一个用户程序的执行自始至终是在操作系统控制下进行的。一个用户使用程序设计语言（比如C语言、Python语言）编写了一个应用程序后，就将该程序连同对它执行的要求输入到计算机内，操作系统就根据要求控制这个用户程序的执行直到结束。操作系统控制用户的执行主要有以下一些内容：

（1）调入相应的编译程序，将使用程序设计语言所编写的源程序编译成计算机可执行的目标代码。

（2）分配内存等资源，然后将程序调入内存并启动程序的执行。

（3）按用户指定的要求处理执行中出现的各种事件以及与操作员联系请示有关意外事件的处理等。

#### 3．人机交互

操作系统的人机交互功能是决定计算机系统友好性的一个重要因素。人机交互功能主要靠可输入/输出的外部设备和相应的软件来完成。可供人机交互使用的设备主要有键盘、显示器、鼠标，以及各种模式识别设备等。与这些设备对应的软件就是操作系统所提供的人机交互功能。人机交互的主要作用是控制有关设备的运行和理解并执行通过人机交互设备传来的各种指令和要求。

#### 4．进程管理

不管是常驻程序还是应用程序，他们都是以进程为标准的执行单位。进程就是当前正在运行的程序。在早期使用冯．诺依曼理论构建计算机系统时，每个中央处理器最多只能同时执行一个进程。早期的操作系统（如[DOS](http://baike.baidu.com/view/365.htm)）不允许任何程序突破这个限制，且它同时只能执行一个进程。而现代的操作系统（如Windows），即便只有一个[CPU](http://baike.baidu.com/view/2089.htm)，但是它也可以利用多任务功能同时执行多个进程。进程管理指的是操作系统管理多个进程的准备、运行、挂起和退出。

由于绝大多数的计算机系统只有一个CPU，在[单内核](http://baike.baidu.com/view/2375980.htm" \t "_blank)CPU的情况下多进程只是简单迅速地切换各进程，以便CPU都能够执行每个进程；而在多内核/多处理器的情况下，通过许多协同技术，在不同的处理器或内核上切换所有进程。很明显，所需要执行的进程越多，每个进程能分配到的时间片就越小。进程管理通常使用分时复用的调度机制，大部分的操作系统可以通过为不同进程指定不同的优先级，从而改变为这些进程所分配的时间片。在进程管理中，优先调度优先级高的进程。

#### 5．内存管理

程序员通常希望系统给进程分配尽可能多且尽可能快的存储器资源。大部分的现代计算机存储器架构都是分层的，存 储器层次按下面顺序排列，即寄存器、高速缓存、内存和外存。寄存器容量最少，而外存容量最多；寄存器速度最快，而外存速度最慢。而操作系统的存储器管理功能提供：

（1）查找可用的存储空间；

（2）配置与释放存储空间；

（3）交换内存和外存的内容；

（4）提供存储器访问的权限等功能。

#### 6．虚拟内存

[虚拟内存](http://baike.baidu.com/view/976.htm)是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间）。而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时保存在外部磁盘存储器上，在需要时进行数据交换。

#### 7．用户接口

用户接口包括作业一级接口和程序一级接口。作业一级接口为了便于用户直接或间接地控制自己的作业而设置。它通常包括联机用户接口与脱机用户接口。程序一级接口是为用户程序在执行中访问系统资源而设置的，通常由一组系统调用组成。

在早期的单用户单任务操作系统中，每台计算机只有一个用户，每次运行一个程序，且程序不是很大，单个程序完全可以存放在实际内存中。这时虚拟内存并没有太大的用处。但随着程序占用存储器容量的增加和以及用户多任务操作系统的出现，程序所需要的存储器资源与计算机系统实际配置的主存储器的容量之间往往存在着矛盾，例如在某些计算机系统中，所提供的物理内存容量较小，而某些应用程序却需要很大的内存空间才能运行；而在多用户多任务系统中，多个用户或多个任务更新全部主存，要求同时互斥（排他性）执行程序。这些同时运行的程序到底占用实际内存中的哪一部分，在编写程序时是无法预先确定的，必须等到运行程序时才能进行分配（动态分配）。

#### 8．用户界面

用户界面（user interface，UI）是系统和用户之间进行交互和信息交换的媒介，它实现信息的内部形式与人类可以接受形式之间的转换。

用户界面是介于用户与硬件而设计彼此之间交互沟通而设计相关软件，目的在使得用户能够方便高效地去操作硬件以实现双向交互，完成借助硬件才能完成的工作，用户界面定义广泛，包含了人机交互与图形用户接口，凡参与人类与机械的信息交流的领域都存在着用户界面。用户和系统之间一般用面向问题的受限自然语言进行交互。目前有系统开始利用多媒体技术开发新一代的用户界面。

### 18.2.3 嵌入式操作系统

嵌入式操作系统，顾名思义就是该操作系统用于嵌入式系统中。那用于嵌入式系统的操作系统和用于计算机的操作系统又有什么不同之处呢？

以STM32G071 MCU为例，该微控制器使用的Arm Cortex-M0+处理器性能根本无法和高性能处理器相比，提供的片上SRAM和Flash存储器资源也非常有限。此外，为了适用于嵌入式的应用场景对MCU的功耗也提出了苛刻的要求。总之一句话，就是MCU上的各种资源非常有限、但是还要求有较好的系统性能，以及整体功耗。

因为有了这个先决条件，所以搭载在STM32G071 MCU的操作系统，必须“小型易用“。所谓的”小型”就是对操作系统进行“裁剪”，使之能充分高效的利用MCU内现有的片上资源，这种“裁剪”仍然要保留操作系统的必要组件。所谓的“易用”就是指应用程序开发人员很容易在”裸机“程序的基础上通过加载和配置嵌入式操作系统，来改造原来的”裸机“程序，使其能更好的满足不同嵌入式应用场景的需求。

典型的，在STM32G071上可以搭载下面的操作系统。

#### 1．μC/OS

MicroC/OS（也称为μC/OS）内核最初发表在《嵌入式系统编程》杂志的三篇文章和Jean J. Labrosse所著的《μC/OS The Real-Time Kernel》（ISBN：0-87930-444-8）一书中。作者最初打算简单描述他为自己开发的便携式操作系统的内部结构，但后来将操作系统开发为版本II和III。

基于为μC/OS编写的源代码，并于1998年作为商业版本推出，μC/OS-II是一个可移植的、可ROM的、可扩展的、抢占式的、实时的、确定性的、多任务内核，用于微处理器和数字信号处理器（digital signal processors，DSPs）。它管理最多255个应用程序任务。它的大小可以进行调整（5~24KB），以仅包含给定用途所需的功能。

大部分的μC/OS-II是用高度可移植的ANSI C编写的，目标微处理器特定的代码是用汇编语言编写的。后者的使用被最小化，以方便的移植到其他处理器。

μC/OS-III是Micro-Controller Operating System Version 3的首字母缩写，于2009年推出，为μC/OS-II RTOST添加了功能。

μC/OS-III提供了μC/OS-II的所有特性和功能。最大的区别在于支持的任务个数。μC/OS-II仅允许255个优先级的每个任务1个，最多255个任务。μC/OS-III允许任意数量的应用程序任务、优先级和每级任务，仅受处理器对存储器的访问限制。

μC/OS-II和μC/OS-III目前由Silicon Labs的子公司Micrium, Inc.维护，可以按产品或按产品线获得许可，读者可以登录https://www.silabs.com/developers/micrium查看相关的资料。

μC/OS-II是为嵌入式使用而设计的。如果生产商有合适的工具链（即C编译器、汇编器和链接器定位器），则可以将μC/OS-II作为产品的一部分嵌入。μC/OS-II用于许多嵌入式系统，包括航空电子设置、医疗设备和装置、数据通信设备、白色家电（电器）、移动电话/个人数字助手（personal digital assistants，PDAs）、工业控制、消费类电子产品，以及汽车。

#### 2．FreeRTOS

FreeRTOS内核最初由Richard Barry于2003年左右开发，后来由Barry的公司Real Time Engineers Ltd开发和维护。2017年，Real Time Engineers Ltd.将FreeRTOS项目的管理权交给了亚马逊网络服务。作为AWS团队的一员，Barry继续在FreeRTOS上工作。

FreeRTOS设计的小巧而简单。内核本身仅包含三个C文件。为了使代码可读、易于移植和可维护，它主要用C编写，但在需要的地方包含了一些汇编函数（主要在特定于架构的调度器程序中）。

FreeRTOS为多线程或任务、互斥、信号量和软件定时器提供了方法。为低功耗应用提供了无滴答模式。支持线程优先级。可以完全静态分配FreeRTOS应用程序。或者，可以使用五中存储器分配方案动态分配RTOS对象，包括：

（1）只分配；

（2）使用非常简单、快速的算法分配和释放；

（3）一种更复杂但快速的分配和释放算法，具有存储器合并；

（4）更复杂方案的替代方案，包括存储器合并，允许将一个堆分解为跨多个存储器区域。

（5）带有一些互斥保护的C库分配和释放。

在FreeRTOS没有Linux或Windows等操作系统中通常提供的高级功能，如设备驱动、高级存储器管理、用户账户和网络。**FreeRTOS可看作是“线程库“而不是”操作系统“**，尽管可以使用命令行界面和类似POSIX的I/O抽象附加组件。

FreeRTOS通过让主机程序以固定的短时间间隔调用线程滴答方法来实现多线程。线程滴答方法根据优先级和循环调度方案切换任务。通常的间隔是1~10mS(1/1000~1/100秒)，通过来自硬件定时器的中断，但是经常改变这个间隔以适应特定应用。

下载包含为每个端口和编译器准比的配置和演示，允许快速应用程序设计。读者可以登录https://www.freertos.org以获取文档和RTOS教程，以及RTOS设计的详细信息。

#### 3．Mbed

Mbed是平台和操作系统，用于基于32位Arm Cortex-M微控制器的互联网连接设备。该类设备也称为物联网（internet of things，IoT）设备。该项目由Arm及其技术合作伙伴共同开发。

可以使用Mbed在线IDE（一个免费的在线代码编辑器和编译器）开发Mbed平台应用程序。只需要在本地PC上安装Web浏览器，因为工程是在云端编译的，即在远程服务器上，使用ARMCC C/C++编译器。Mbed IDE为私有工作区提供了导入、导出和与分布式Mercurial版本控制共享代码的能力，它还可以用于生成代码文档。开可以使用其他开发环境开发应用程序，例如Keil μVision、IAR Embedded Workbench和带有GCC Arm Embedded工具的Eclipse。

Mbed OS提供Mbed C/C++软件平台和工具，用于创建在IoT设备上运行的微控制器固件。它由提供微控制器外外设驱动程序、网络、RTOS和运行时环境、编译工具以及测试和调试脚本的核心库构成。这些连接可以通过兼容的SSL/TLS（例如支持mbed-rtos的Mbed TLS或wolfSSL）来保护。

元件数据库为用于构建最终产品的元件和服务提供驱动程序库。

Mbed OS，即RTOS，基于Keil RTX5。

#### 4．RT-Thread

RT-Thread成立于2006年，是一个主要用C语言编写的开源实时操作系统（RTOS）。RT-Thread，全称是Real Time-Thread，表示它是一个嵌入式实时多线程操作系统，它的基本属性就是支持多任务，任务调度器根据优先级在不同的任务之间进行快速切换，因此使得用户看起来好像处理器在同一时刻执行了多个任务。在RT-Thread系统中，任务通过线程实现。读者可以通过登录下面的网址https://www.rt-thread.org/获取关于该操作系统的更多信息。

RT-Thread主要使用C语言编写，浅显易懂，方便移植。它把面向对象的设计方法应用到实时系统设计中，使得代码风格优雅、结构清晰、系统模块化且裁剪性非常好。特别是对于资源有限的MCU，可通过使用不同的软件工具，裁剪出需要3KB Flash存储器、1.2KB RAM存储器的NANO版本（该版本是RT-Thread官方于2017年7月发布的一个极简版内核）。而对于资源丰富的物联网设备，RT-Thread又能使用在线的软件包管理工具，配合系统配置工具实现直观快速的模块化裁剪，无缝的导入丰富的软件功能包，实现类似安卓（Android）的图形界面及触摸滑动效果、智能语音交互效果等复杂功能。

RT-Thread系统完全开源，3.1.0以及以前的版本遵守GPL V2+开源协议。3.1.0后的版本遵守Apache License 2.0开源许可协议，可以免费在商业产品中使用，并且不需要公开私有代码。

## 18.3 RT-Thread Nano架构及功能

RT-Thread Nano是一个极简版的硬实时内核，它由C语言开发，采用面向对象的编程思想，具有良好的代码风格，是一个可裁剪的、抢占式实时多任务RTOS。该操作系统所占用的存储器资源很少。该操作系统提供的功能包括任务处理、软件定时器、信号量、邮箱和实时调度等相对完整的实时操作系统特性。该操作系统适用于家电、消费电子、医疗设备、工控等领域大量使用32位Arm入门级MCU的场合。

RT-Thread Nano的内部结构，如图18.1所示。

同步与通信

信号量

事件集

互斥量

邮箱

消息队列

线程管理

时钟管理

中断管理

内存管理

FinSH组件

Arm

RISC-V

…

可移植组件

硬实时内核

CPU架构

图18.1 RT-Thread Nano内部结构

注：支持的Arm架构包括Cortex-M0、Cortex-M0+、Cortex-M3、Cortex-M4和Cortex-M7等。

RT-Thread Nano的特点包括：

（1）下载简单。RT-Thread Nano以软件包的形式集成在STM32CubeMX和Keil μVision开发环境中，可以直接在软件中下载Nano软件包获取源码。同时也提供下载Nano源码压缩包的途径，方便在其他开发环境移植RT-Thread Nano，比如在IAR上移植RT-Thread Nano。

（2）代码简单。与RT-Thread完整板不同的是，Nano不含Scons构建系统，不需要Kconfig以及Env配置工具，也去掉了完整版特有的device框架和组件，仅是一个纯净的内核。

（3）移植简单。由于Nano的极简特性，使Nano的移植过程变得非常简单。添加Nano源码到工程，就完成了90%以上的移植工作。

在Keil μVision与STM32CubeMX开发环境中还提供了Nano软件包，可以一键下载到工程。另外，在RT-Thread Studio中可以基于Nano创建工程直接使用。

（4）使用简单。RT-Thread Nano在使用上也非常简单，带给嵌入式系统开发人员更好的体验。包括：

易裁剪：Nano的配置文件为rtconfig.h，该文件列出了内核中的所有宏定义。

易添加FinSH组件。可以很方便的在Nano上移植FinSH组件，而不依赖于device框架，只需要对接两个必要的函数就可以完成移植。

自选驱动库。可以使用厂商提供的固件驱动库，如ST的STD库、HAL库和LL库。

完善的文档。包含内核基础、线程管理（例程）、时钟管理（例程）、线程间同步（例程）、线程间通信（例程）、内存管理（例程）、中断管理，以及Nano的移植教程。

（5）资源占用少。对RAM和ROM的开销很小，在支持信号量（semaphore）和邮箱特性，并运行两个线程（main线程+idle线程）情况下，仍然保持占用很少的ROM和RAM资源。

（6）开源免费（Apache 2.0）。RT-Thread Nano实时操作系统遵循Apache许可证2.0版本，可以免费在商业产品中使用实时操作系统内核以及所有开源组件，不需要公布应用程序代码，没有潜在风险。

## 18.4 RT-Thread Nano在Keil MDK的移植

本节将介绍在Keil μVision集成开发环境中移植RT-Thread Nano的方法，在移植的过程中使用了STM32G071 MCU器件以及STM32CubeMX生成的Keil MDK工程。

### 18.4.1 安装RT-Thread Nano

本节将介绍在Keil μVision中安装RT-Thread Nano的方法。主要步骤包括：

（1）启动Keil μVision集成开发环境（以下简称Keil）。

（2）在Keil主界面主菜单下，选择Project->Open Project。

（3）弹出Select Project File对话框界面。在该界面中，将路径指向E:\STM32G0\_example\example\_18\_1\MDK-ARM。在该路径下，找到并选中LED\_Control.uvprojx工程文件。

（4）单击打开按钮，退出Select Project File对话框界面。

（5）在Keil左侧的Project窗口中，找到并双击main.c文件。注意到，在while循环中使用了下面的代码来控制开发板上LED灯的变化，如代码清单18-1所示。

代码清单18-1 main.c文件中的main()函数代码

void SystemClock\_Config(void);

static void MX\_GPIO\_Init(void);

int main(void)

{

HAL\_Init(); //复位所有外设、初始化Flash和Systick

SystemClock\_Config(); //配置系统时钟

MX\_GPIO\_Init(); //初始化所有配置的外设

while (1) //while循环

{

HAL\_GPIO\_TogglePin(GPIOA,GPIO\_PIN\_5); //切换GPIO的状态

HAL\_Delay(500); //延迟

}

}

（6）在Keil主界面的工具栏中单击Pack Installer按钮，如图18.2所示。

（7）弹出新的Pack Installer页面，以及Pack Installer对话框界面。在对话框界面中，提示Welcome to the Keil Pack Installer信息，单击OK按钮退出该对话框界面。

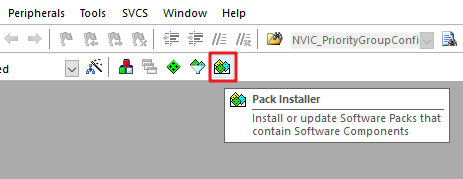


图18.2 Pack Installer入口

（8）在Pack Installer页面的右侧窗口中，单击Packs标签，如图18.3所示。在该标签界面中，找到并展开RealThread::RT-Thread条项。单击该条项右侧的Install按钮。

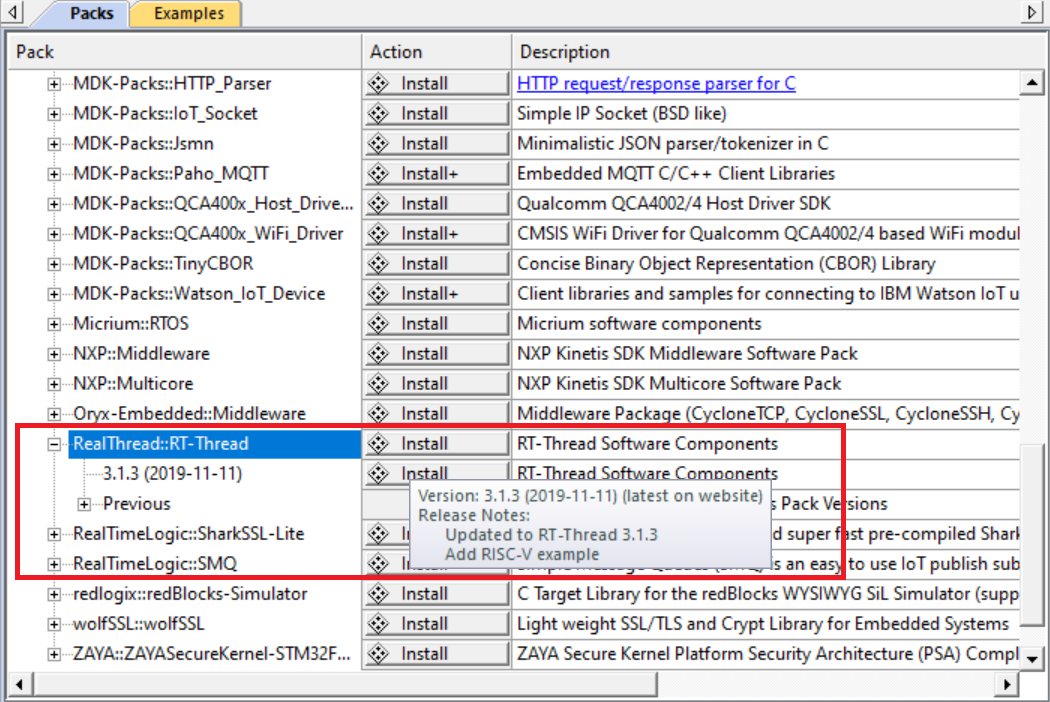


图18.3 RT-Thread的安装入口

（9）弹出Pack Unzip: RealThread RT-Thread 3.1.3-License Agreement对话框界面。在该界面中，勾选I agree to all the terms of the preceding License Agreement前面的复选框。

（10）单击Next按钮，下载并安装RT-Thread。

（11）退出Pack Installer页面。

### 18.4.2 添加RT-Thread Nano

本节将RT-Thread Nano添加到当前的设计工程中，主要步骤包括：

（1）在Keil主界面工具栏中，找到并单击Manage Run-Time Environment按钮，如图18.4所示。

（2）弹出Manage Run-Time Environment页面，如图18.5所示。在该页面中，找到RTOS条项，通过该条项右侧的下拉框将其设置为RT-Thread，表示RTOS使用的是RT-Thread。

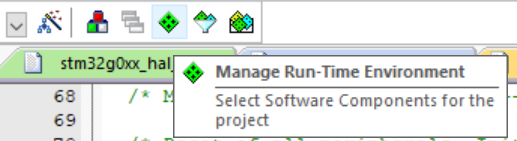


图18.4 Manage Run-Time Environment按钮

展开RTOS条项。在展开项中，勾选kernel右侧的复选框。

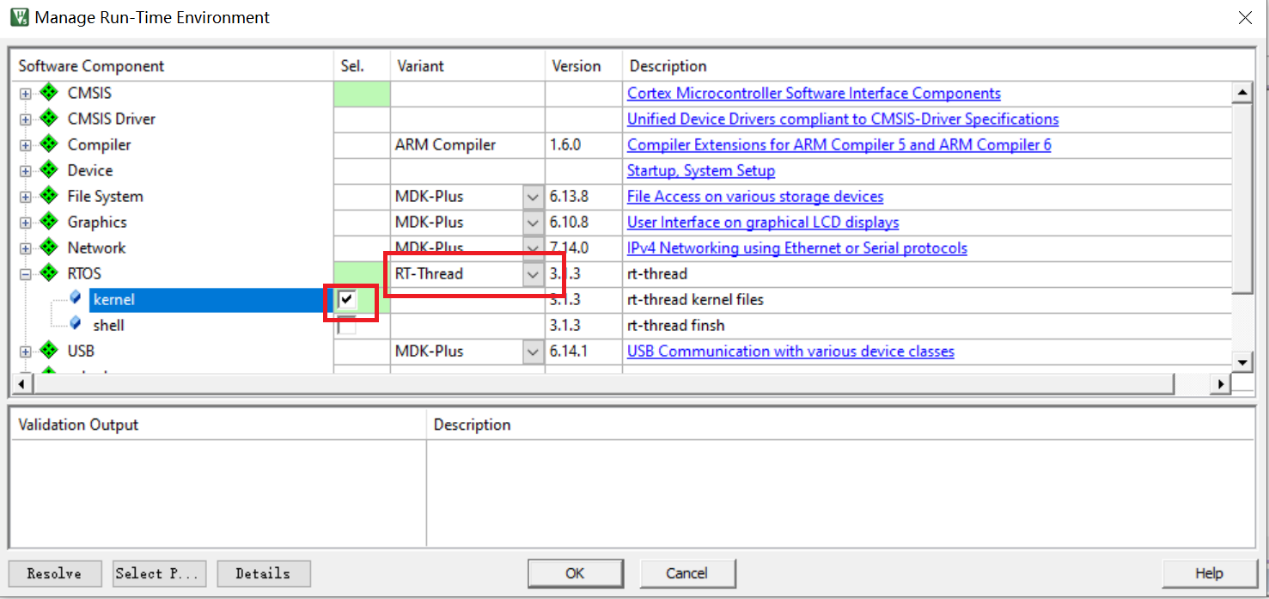


图18.5 将RT-Thread添加到当前的工程中

（3）在Keil主界面左侧的Project窗口中，找到并展开RTOS条项，如图18.6所示。很

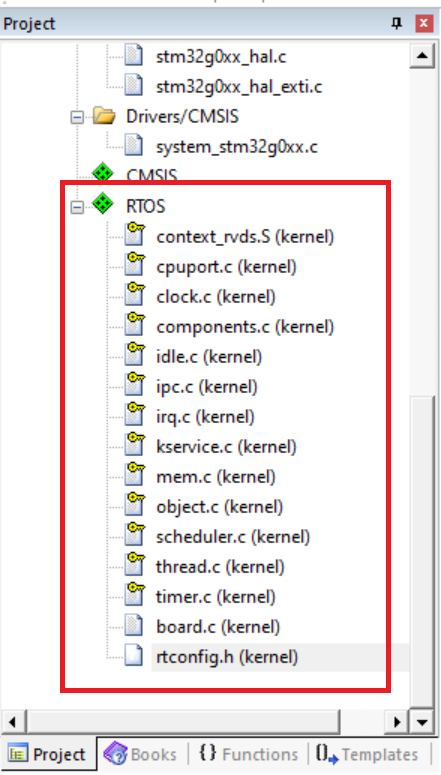


图18.6 添加到当前工程中的RT-Thread文件

明显，RT-Thread中的文件已经添加到当前工程中。其中：

Cortex-M芯片内核移植代码文件，包括context\_rvds.s和cpuport.c文件。

Kernel文件，包括clock.c、components.c、device.c、idle.c、ipc.c、irq.c、kservice.c、mem.c、object.c、scheduler.c、thread.c和time.c。

配置文件，包括board.c和rtconfig.h。

### 18.4.3 适配RT-Thread Nano

本节需要修改工程中的一些文件，使得在当前的工程中可以正确的加载RT-Thread。

#### 1．修改中断与异常处理

RT-Thread会接管异常处理函数HardFault\_Handler()和挂起处理函数PendSV\_Handler()，这两个函数已由RT-Thread实现**，所以需要删除工程里中断服务例程文件中的这两个函数，避免在编译时产生重复定义的错误。**

方法是：在Keil主界面左侧的Project窗口中，找到并单击stm32g0xx\_it.c文件。找到异常处理函数HardFault\_Handler()和挂起处理函数PendSV\_Handler()的定义部分，使用注释符号将这两个函数注释掉。

注：如果此时对工程进行编译，没有出现函数重复定义的错误，则不用进行任何修改。

#### 2．配置系统时钟

需要在board.c文件中实现系统时钟配置（为MCU、外设提供工作时钟）与OS tick（为操作系统提供节拍）。如代码清单18-2所示。

代码清单18-2 rt\_hw\_board\_init()函数

void rt\_hw\_board\_init()

{

   HAL\_Init();                                 //复位所有外设、初始化Flash和Systick

   SystemClock\_Config();                     //配置系统时钟

//更新系统时钟

SystemCoreClockUpdate();

//配置系统滴答时钟

\_SysTick\_Config(SystemCoreClock / RT\_TICK\_PER\_SECOND);

/\* 调用元件板初始化(use INIT\_BOARD\_EXPORT()) \*/

#ifdef RT\_USING\_COMPONENTS\_INIT

rt\_components\_board\_init();

#endif

#if defined(RT\_USING\_USER\_MAIN) && defined(RT\_USING\_HEAP)

rt\_system\_heap\_init(rt\_heap\_begin\_get(), rt\_heap\_end\_get());

#endif

}

此外，使用滴答定时器systick实现OS Tick，在board.c中实现SysTick\_Handler()中断服务例程，调用RT-Thread提供的rt\_tick\_increase()函数，如代码清单18-3所示。

代码清单18-3 SysTick\_Handler()函数

void SysTick\_Handler(void)

{

/\* enter interrupt \*/

rt\_interrupt\_enter();

rt\_tick\_increase();

/\* leave interrupt \*/

rt\_interrupt\_leave();

}

由于在board.c中重新实现了SysTick\_Handler()中断服务例程，做了OS Tick，所以需要**删除工程里原来已经实现的SysTick\_Handler()**，避免在编译时产生重复定义的错误。方法是：在Keil主界面左侧的Project窗口中，找到并单击stm32g0xx\_it.c文件。找到异常处理函数SysTick\_Handler()定义部分，使用注释符号将这个函数注释掉。

#### 3．初始化堆

在board.c文件的rt\_hw\_board\_init()函数中，提供了初始化系统存储器堆（heap）的功能。通过使能/禁止RT\_USING\_HEAP，来确定是否使用存储器堆功能，RT-Thread Nano默认不使能存储器堆功能，这样可以保持较少的存储器资源开销，不用为存储器堆开辟空间。

开启系统堆将可以使用动态存储器分配功能，如使用rt\_malloc、rt\_free以及各种系统动态创建对象的API。如果需要使用系统存储器堆功能，则使能RT\_USING\_HEAP，此时将调用存储器初始化函数rt\_system\_heap\_init()，如代码清单18-2所示。

初始化存储器堆需要堆的起使地址和结束地址两个参数，系统中默认使用数组作为堆，并获取了堆的起始地址和结束地址，该数据的大小可修改。在board.c中找到该段代码，如代码清单18-4所示。

代码清单18-4 堆的定义

#if defined(RT\_USING\_USER\_MAIN) && defined(RT\_USING\_HEAP)

#define RT\_HEAP\_SIZE 1024

static uint32\_t rt\_heap[RT\_HEAP\_SIZE]; // heap default size: 4K(1024 \* 4)

RT\_WEAK void \*rt\_heap\_begin\_get(void)

{

return rt\_heap;

}

RT\_WEAK void \*rt\_heap\_end\_get(void)

{

return rt\_heap + RT\_HEAP\_SIZE;

}

#endif

**需要注意，使能堆的存储器动态分配后，堆的默认值较小，在使用的时候需要增加堆的大小，否则会有申请存储器失败或者创建线程失败的情况。提供下面两种修改方法：**

**（1）可以直接修改数组中定义的RT\_HEAP\_SIZE的值，至少大于各个动态申请的存储器容量之和，但是要小于所能提供的RAM容量。**

**（2）使用RAM ZI段结尾处作为HEAP的起始地址，使用RAM的结尾地址作为HEAP的结尾地址，这是heap能设置的最大值的方法。**

### 18.4.4 修改设计代码

本节将修改初始的main.c设计文件，主要步骤包括：

（1）打开main.c文件，添加与RT-Thread相关的头文件

#include "rtthread.h"

（2）在while循环中将延迟函数改为rt\_thread\_mdelay()。该函数会引起系统调度，切换到其他线程运行，体现了线性实时性的特点，如代码清单18-5所示。

代码清单18-5 while循环语句

while (1)

{

HAL\_GPIO\_TogglePin(GPIOA,GPIO\_PIN\_5);

rt\_thread\_mdelay(500);

}

（3）保存设计文件。

（4）在Keil主界面主菜单中，选择Project->Build Target，对设计进行编译和链接，生成可下载到STM32G071 MCU片内Flash存储器的文件格式。

（5）通过USB电缆，将计算机/笔记本电脑的USB接口连接到NUCLEO-G071RB开发板的USB接口。

（6）在Keil主界面主菜单中，选择Flash->Download，将生成的存储器文件格式的代码下载到STM32G071 MCU片内的Flash存储器中。

（7）按一下开发板上标记为RESET按键，使得程序正常运行。

思考与练习18-1：观察LED灯的状态，所添加的RT-Thread函数是否起作用。

注：当添加RT-Thread之后，原来工程中的main()函数会自动成为RT-Thread系统中的main线程的入口函数。由于线程不能一直独占CPU，所以此时在main()中使用while(1)时，需要有让出CPU的动作，比如使用rt\_thread\_mdelay()则可以让出CPU资源。

### 18.4.5 配置RT-Thread Nano

RT-Thread Nano默认为使能RT\_USING\_HEAP，因此只支持静态方式创建任务、信号量等对象。若要通过动态方式创建对象则需要在rtconfig.h文件中使能RT\_USING\_HEAP宏定义。

在图18.6中找到并双击rtconfig.h，打开配置向导（Configuration Wizard）界面，如图18.7所示。展开Memory Management Configuration条项。在展开项中，可以看到勾选/不勾选Dynamic Heap Management使能/禁止动态堆管理。在界面中所做的修改就等同于直接修改rtconfig.h文件。

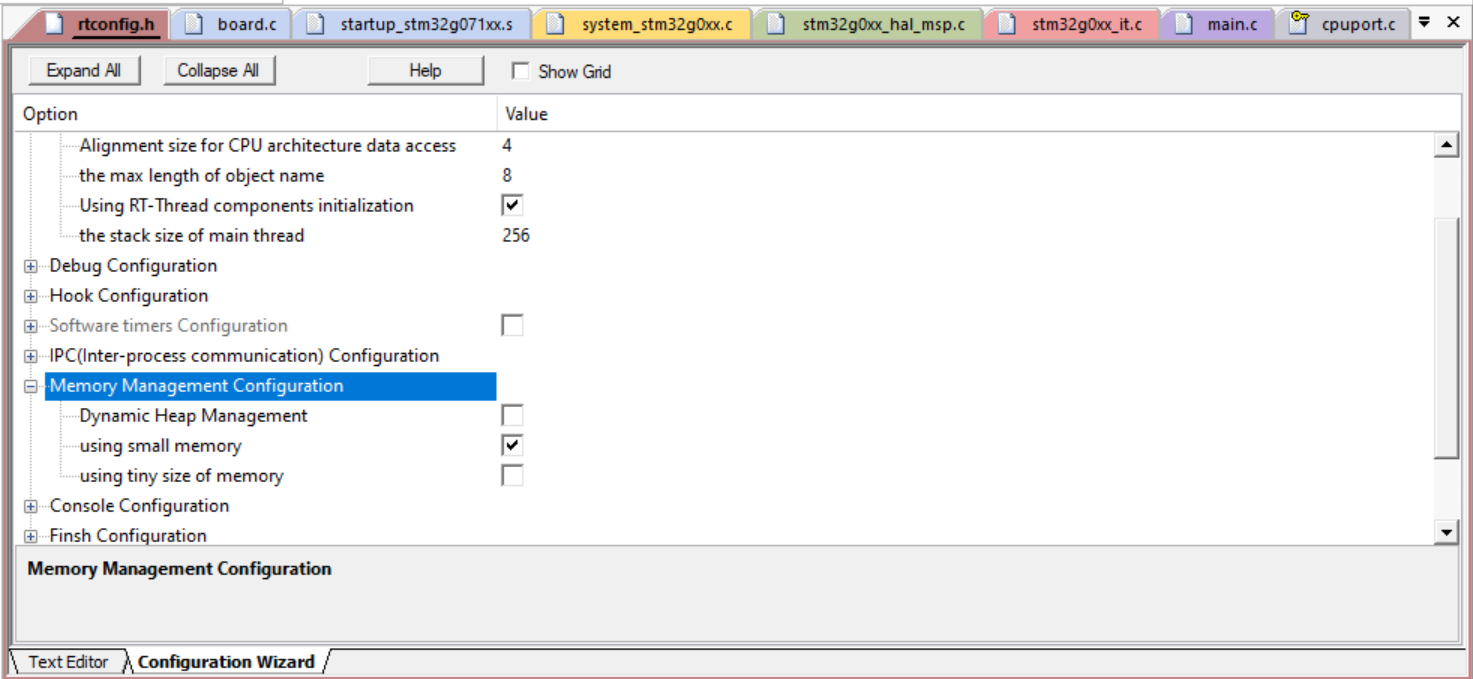


图18.7 Configuration Wizard中提供的配置选项

#### 1．基础配置

（1）设置操作系统的最大优先级，可设置范围8~256，默认值为8，可修改。

#define RT\_THREAD\_PRIORITY\_MAX 8

（2）设置RT-Thread操作系统的节拍，表示每秒的节拍数，如默认值为1000，表示一个时钟节拍（os tick）长度为1ms。时钟节拍频率越快，操作系统的额外开销就越大。

#define RT\_TICK\_PER\_SECOND 1000

（3）字节对齐时设置对齐的字节个数，默认4，常使用ALIGN(RT\_ALIGN\_SIZE)进行字节对齐。

#define RT\_ALIGN\_SIZE 4

（4）设置对象名字的最大长度，默认8个字符，一般无需修改。

#define RT\_NAME\_MAX 8

（5）设置使用组件自动初始化功能，默认需要使用，使能该宏则可以使用自动初始化功能。

#define RT\_USING\_COMPONENTS\_INIT

（6）使能RT\_USING\_USER\_MAIN宏，则打开user\_main功能，默认需要使能，这样才能调用RT\_Thread的启动代码；main线程的堆栈大小默认为256，可修改。

#define RT\_USING\_USER\_MAIN

#define RT\_MAIN\_THREAD\_STACK\_SIZE 256

#### 2．内核调试功能配置

定义RT\_DEBUG宏则使能debug模式，默认不使能。若使能系统调试，则可以打印系统LOG日志。

//#define RT\_DEBUG //关闭debug

#define RT\_DEBUF\_INIT 0 //使能组件初始化调试配置，设置为1则//会打印自动初始化的函数名字

//define RT\_USING\_OVERFLOW\_CHECK //禁止栈溢出检查

#### 3．挂钩函数

设置是否使用挂钩函数，默认禁止。

//#define RT\_USING\_HOOK //是否使能系统挂钩函数

//#define RT\_USING\_IDLE\_HOOK //是否使能空闲线程挂钩函数

#### 4．软件定时器配置

设置是否使能软件定时器，以及相关参数的配置，默认禁止。

#define RT\_USING\_TIMER\_SOFT 0 //关闭软件定时器功能，为1则打开

#if RT\_USING\_TIMER\_SOFT == 0

#undef RT\_USING\_TIMER\_SOFT

#endif

#define RT\_TIMER\_THREAD\_PRIO 4 //设置软件定时器线程优先级，默//认4

#define RT\_TIMER\_THREAD\_STACK\_SIZE 512 //设置软件定时器堆栈大小，默认//512字节

#### 5．IPC配置

系统支持的进程间通信（inter-process communication，IPC）包括信号量、互斥、事件集、邮箱、消息队列。通过定义相应的宏使能/禁止使用该IPC。

//define RT\_USING\_SEMAPHORE //设置是否使用信号量

//define RT\_USING\_MUTEX //设置是否使用互斥

//define RT\_USING\_EVENT //设置是否使用事件集

//define RT\_USING\_MAILBOX //设置是否使用邮箱

//define RT\_USING\_MESSAGEQUEUE //设置是否使用消息队列

#### 6．存储器配置

RT\_Thread存储器管理包括存储器池、存储器堆、小存储器算法。通过使能相应的宏定义使用相应的功能。

//define RT\_USING\_MEMPOOL //是否使用存储器池

//define RT\_USING\_HEAP //是否使用存储器堆

//define RT\_USING\_SMALL\_MEM //是否使用小存储器管理

//define RT\_USING\_TINY\_SIZE //是否使用小体积的算法

#### 7．FinSH控制台配置

定义RT\_USING\_CONSOLE则使能控制台功能，使能该宏则关闭控制台，不能实现打印；修改RT\_CONSOLEBUF\_SIZE可配置控制台缓冲大小。

//define RT\_USING\_CONSOLE //控制台宏开关

//define RT\_CONSOLEBUF\_SIZE 128 //设置控制台数据缓冲区大小，默认128B

通过定义RT\_USING\_FINSH使能使用FinSH组件，使能后可对FinSH组件的相关参数进行配置修改，FINSH\_THREAD\_STACK\_SIZE的默认值较小，根据实际情况增加该值。

#if defined(RT\_USING\_FINSH) //开关FinSH组件

#define FINSH\_USING\_MSH //使用FinSH组件MSH模式

#define FINSH\_USING\_MSH\_ONLY //仅使用MSH模式

#define \_\_FINISH\_THREAD\_PRIORITY 5 //设置FinSH优先级，配置后根据下面公式进行计算

#define FINSH\_THREAD\_PRIORITY (RT\_THREAD\_PRIORITY\_MAX / 8 \* \_\_FINSH\_THREAD\_PRIORITY + 1)

#define FINSH\_THREAD\_STACK\_SIZE 512 //设置FinSH线程栈大小，范围1~4096

#define FINSH\_HISTORY\_LINES 1 //设置FinSH组件记录历史命令的个数，值的范围1~32

#define FINSH\_USING\_SYMTAB //使用符号表，需要使能，默认使能

#endif

## 18.5 RT-Thread Nano内核分析与实现

本节将对RT-Thread Nano内核的关键部分进行说明，并通过设计实例进行演示。

注：（1）该部分内容参考了RT-Thread的文档和设计案例，读者可以登录下面的网址https://www.rt-thread.org/查找这些资料和设计案例。

（2）由于RT-Thead Nano的内核功能较多，本书仅以线程创建和调用、定时器调用，以及线程同步为例，来举例说明内核的功能，更多的设计案例可以参考RT-Thread提供的实验材料。

### 18.5.1 线程及其管理

线程是RT-Thread操作系统中最小的调度单位，线程调度算法是基于优先级的全抢占式多线程调度算法，即在系统中除了中断处理函数、调度器上锁部分的代码和禁止中断的代码时不可抢占的之外，系统的其他部分都是可以抢占的，包括线程调度器自身。RT-Thread支持256个线程优先级（可以通过配置文件更改为支持32个/8个线程优先级，针对STM32的默认配置时32个线程优先级）。0优先级表示最高优先级，最低优先级留给空闲线程使用；同时RT-Thread也支持创建多个具有相同优先级的线程，相同优先级的线程间采用时间片的轮询调度算法进行调度，使每个线程运行相应时间；另外，调度器在寻找那些处于就绪状态的具有最高优先级的线程时，所经历的时间是恒定的，系统也不限制线程数量的多少，线程数目只与硬件的具体存储器相关。

在日常生活中，我们经常会把一个大的复杂的任务分解为一个个小的比较容易实现的任务，然后按一定的规则将小的任务一个个实现，这样就可以完成一个大的复杂的任务。在计算机软件程序设计中，我们将一个代码长度较长，实现起来比较复杂的程序，分解为一段一段的小的比较容易实现的代码，然后按一定的规则来实现这些小的容易实现的代码。

比如对于一个任务，该任务通过传感器采集数据，然后在显示器上将采集的数据显示出来。我们可以将这个任务分解为两个子任务，一个子任务不间断的读取传感器的数据，并将数据写到存储器中；另一个子任务周期性的从存储器中读取数据，并将传感器数据输出到显示屏上，如图18.8所示。

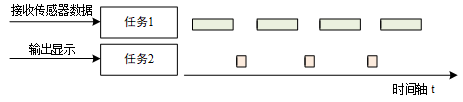


图18.8 数据采集和显示任务的划分和处理

在RT-Thread中，与上述子任务对应的程序实体就是线程，线程是实现任务的载体，它是RT-Thread中最基本的调度单位，它描述了一个任务执行的运行环境，也描述了这个任务所处的有限等级，可以将紧急任务设置较高的优先级，将非紧急任务设置低优先级。当然，不同的任务也可以设置相同的优先级，轮流运行。

对于每个运行的线程，它本身会认为自己独占了CPU的资源。运行线程时的环境称为**上下文**，正确的上下文使得不同的线程可以根据调度策略进行切换。

在rtdef.h文件中，使用struct rt\_thread表示线程控制块，具体见该文件。该线程控制块是操作系统中用于管理线程的一个数据结构，它会存放线程的一些信息，例如优先级、线程名字、线程状态等，也包含线程与线程之间连接用的链表结构，线程等待事件集合等。

RT-Thread线程具有独立的栈，当切换线程时，会将当前线程的上下文保存在堆栈中，当线程要恢复运行时，再从堆栈中读取上下文信息，进行恢复。

在线程运行的过程中，同一时间内只允许一个线程在处理器中运行，从运行的过程上划分，线程有多种不同的运行状态，如初始状态、挂起状态、就绪状态等。在RT-Thread中，线程包含5种状态，如表18.1所示。

表18.1 线程的5种状态

|  |  |
| --- | --- |
| 状态 | 描述 |
| 初始状态 | 当刚创建线程而没有开始运行线程时，该线程就处于初始状态；在初始状态下，线程不参与调度。在状态在RT-Thread中的宏定义为RT-THREAD\_INIT |
| 就绪状态 | 在就绪状态下，线程按照优先级排队，等待执行它；一旦当前线程运行完毕让出处理器时，操作系统会马上寻找最高优先级的就绪态线程运行。在该状态在RT-Thread中的宏定义为RT\_THREAD\_READY |
| 运行状态 | 当前正在运行线程。在单核系统中，只有rt\_thread\_self()函数返回的线程处于运行状态；在多核系统中，可能不止这一个线程处于运行状态。该状态在RT-Thread中的宏定义为RT-THREAD\_RUNNING |
| 挂起状态 | 也称为阻塞状态。这是由于系统未能给线程提供需要的资源，使得线程处于挂起等待状态，或线程主动延迟一段时间而挂起。在挂起状态下，线程不参与调度。该状态在RT-Thread中的宏定义为RT-THREAD-SUSPEND |
| 关闭状态 | 当线程结束运行时，将处于关闭状态。处于关闭状态的线程不参与线程的调度。该状态在RT-Thread中的宏定义为RT-THREAD-CLOSE |

操作系统会自动根据它运行的情况来动态调整它的状态，如图18.9所示。从图中可知：

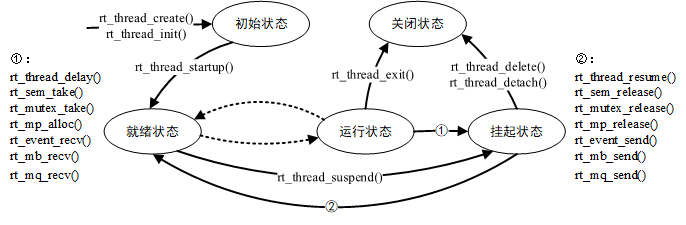


图18.9 线程之间的状态切换

（1）线程通过调用函数rt\_thread\_create/init()进入到初始状态（RT\_THREAD\_INIT）；

（2）初始状态的线程通过调用函数rt\_thread\_startup()进入到就绪状态（RT\_THREAD\_READY）;

（3）就绪状态的线程被调度器调度后进入运行状态（RT\_THREAD\_RUNNING）;

（4）当处于运行状态的线程调用rt\_thread\_delay()、rt\_sem\_take()、rt\_mutex\_take()或rt\_mb\_recv()等函数或获取不到资源时，将进入到挂起状态（RT\_THREAD\_SUSPEND）;

（5）处于挂起状态的线程，如果等待超时仍然没有获得资源或者由于其他线程释放了资源，那么它将返回到就绪状态；

（6）挂起状态的线程，如果调用rt\_thread\_delete/detach()函数，将更改为关闭状态（RT\_THREAD\_CLOSE）;

（7）处于运行状态的线程，如果运行结束，就会在线程的最后部分执行rt\_thread\_exit()函数，将状态更改为关闭状态。

注：在RT-Thread OS中，实际上线程并不存在运行状态，就绪状态和运行状态是等同的。

在系统启动时，系统会创建main线程，它的入口函数为main\_thread\_entry()，用户的应用入口函数main()就是从这里真正开始的，启动系统调度器后，就开始运行main线程，如图18.10所示，用户可以在main()函数里添加自己的应用程序初始化代码。



图18.10 主线程

#### 1．创建线程

一个线程要称为可执行的对象，就必须由操作系统的内核为它创建一个线程，创建函数的原型如下所示：

rt\_thread\_t rt\_thread\_create(const char \*name,

void (\*entry)(void \*parameter),

void \*parameter,

rt\_uint32\_t stack\_size,

rt\_uint8\_t priority,

rt\_uint32\_t tick);

该函数的参数说明，如表18.2所示。

|  |  |
| --- | --- |
| 参数 | 描述 |
| name | 线程的名字；线程名字的最大长度由rtconfig.h中宏RT\_NAME\_MAX指定，超过该长度的名字将被截掉 |
| entry | 线程入口参数 |
| parameter | 线程入口参数 |
| stack\_size | 线程堆栈大小，单位是字节 |
| priority | 线程的优先级。根据系统配置情况确定优先级范围（由rtconfig.h中的RT\_THREAD\_PRIORITY\_MAX定义），如果支持的是256级优先级，那么范围从0~255，数值越小优先级越搞，0代表最高优先级 |
| tick | 线程的时间片大小。时间片（tick）的单位是操作系统的时钟节拍。当系统中存在相同优先级的线程时，该参数指定调度一次线程能够运行的最大时间长度。当该时间片运行结束时，调度器自动选择下一个就绪状态的同优先级的线程进行运行 |

当线程创建成功时，返回thread（线程创建成功，返回线程句柄）；当线程创建失败时，返回RT\_NULL。

#### 2．删除线程

对于一些使用rt\_thread\_create()创建出来的线程，当不需要使用时，或者运行出错时，可以使用下面的函数删除该线程，

rt\_err\_t rt\_thread\_delete(rt\_thread\_t thread);

调用该函数后，将该线程移出线程队列并从内核对象管理器中删除，同时也会释放该线程所占用的堆栈空间，回收的该堆栈空间将重新进行分配以用于其他线程。实际上，用rt\_thread\_delete()函数删除线程，仅仅是把相应的线程状态更改为RT\_THREAD\_CLOSE状态，然后放到rt\_thread\_defunct队列中；而删除线程的真正行为（释放线程块和释放线程栈）需要到下一次执行空闲进程时，由空闲线程完成最后的线程删除动作。

当删除线程成功时，返回RT\_EOK；当删除线程失败时，返回RT\_ERROR。

注：仅在使能系统动态堆时，rt\_thread\_create()和rt\_thread\_delete()函数才有效（即在RT\_USING\_HEAP宏定义中进行了定义）。

#### 3．初始化线程

使用下面的函数完成对线程的初始化，以初始化静态线程对象。

rt\_err\_t rt\_thread\_init(struct rt\_thread \*thread,

const char \*name,

void (\*entry)(void \*parameter),

void \*parameter,

void \*stack\_start,

rt\_uint32\_t stack\_size,

rt\_uint8\_t priority,

rt\_uint32\_t tick);

其中：（1）thread：线程句柄。由用户提供，并指向对应的线程控制块存储器地址。

（2）stack\_start：线程栈的起始地址。

当创建线程成功时，返回RT\_EOK；当创建线程失败时，返回RT\_ERROR。

#### 4．脱离线程

对于使用rt\_thread\_init()初始化的线程，使用rt\_thread\_detach()将脱离在线程队列和内核对象管理器中的该线程。线程脱离函数如下：

rt\_err\_t rt\_thread\_detach(rt\_thread\_t thread);

当成功脱离线程时，返回RT\_EOK；当脱离失败时，返回RT\_ERROR；

#### 5．启动线程

创建（初始化）的线程状态处理初始状态，并未进入就绪线程的调度队列，可以在线程初始化/创建成功后，调用下面的函数让线程进入就绪状态：

rt\_err\_t rt\_thread\_startup(rt\_thread\_t thread);

当调用该函数时，将线程的状态更改为就绪状态，并放到相应优先级队列中等待调度。如果新启动的线程优先级高于当前的线程优先级，则立即切换到新启动的线程。

当成功启动线程时，返回RT\_EOK；当线程启动失败后，返回RT\_ERROR。

#### 6．获得当前的线程

在程序运行的过程中，多个线程可能会执行相同的一段代码，可以通过下面的函数获得当前执行的线程句柄:

rt\_thread\_t rt\_thread\_self(void);

返回值为当前运行的线程句柄，如果返回RT\_NULL，则表示还未启动调度器。

#### 7．让出线程

当前线程的时间片用完或者该线程主动要求让出处理器资源时，它将不再占有处理器，调度器会选择执行具有相同优先级的下一个线程。线程调用该函数后，该线程仍然在就绪队列中。线程使用下面的函数让出处理器：

rt\_err\_t rt\_thread\_yield(void);

调用该函数后，当前线程首先把自己从它所在的就绪优先级队列中删除，然后将自己挂到该优先级队列链表的尾部，然后激活调度器切换上下文（如果当前优先级只有这一个线程，则继续执行该线程，不进行上下文切换）。

rt\_thread\_yield()函数和rt\_schedule()函数类似，但在有相同优先级的其他就绪状态线程存在时，系统的行为却截然不同。执行rt\_thread\_yield()函数后，将当前线程换出，然后执行下一个具有相同优先级的线程。但是在执行rt\_schedule()函数后，并不一定把当前线程换出，即使将其换出，也不会被放到就绪队列线程链表的尾部，而是在系统中选取就绪的优先级最高的线程执行（如果系统中没有存在比当前线程优先级更高的线程，那么执行完rt\_schedule()函数后，系统将继续执行当前线程）。

#### 8．线程休眠

在实际应用中，有时需要让运行的当前线程延迟一段时间，在到达指定的时间后重新运行，这就是线程休眠。使用下面的函数，可以实现线程休眠：

rt\_err\_t rt\_thread\_delay(rt\_tick\_t tick);

rt\_err\_t rt\_thread\_mdelay(rt\_int32\_t ms);

这些函数的作用相同，调用他们可以使当前的线程挂起一段指定的时间，当这个时间过后，将线程唤醒并使其进入就绪状态。该函数接受一个参数，该参数指定了线程的休眠时间。线程休眠函数的入口参数为tick/ms，tick以1个OS Tick为单位，mdelay以1ms为单位。当操作成功后，返回RT\_EOK。

#### 9．挂起线程

挂起线程使用下面的函数：

rt\_err\_t rt\_thread\_suspend(rt\_thread\_t thread);

当线程挂起成功后，返回RT\_EOK；当线程挂起失败后，返回RT\_ERROR，因为该线程的状态并不是就绪状态。

注：只能使用本函数挂起线程自己，不可以在线程A中尝试挂起线程B，而且在挂起线程自己后，需要立即调用rt\_schedule()函数进行手动的线程上下文切换。用户只需要了解该函数的作用，不推荐使用该函数，该函数可看作使内核内部函数。

#### 10．线程恢复

恢复线程就是让挂起的线程重新进入就绪状态，并将线程放入系统的就绪队列中；如果恢复的线程在所有就绪线程中位于最高优先级链表的第一位，那么系统将进行线程的上下文切换，使用下面的函数恢复线程

rt\_err\_t rt\_thread\_resume(rt\_thread\_t thread);

当线程恢复成功时，返回RT\_EOK；当线程恢复失败时，返回RT\_ERROR，因此该线程的状态并不是RT\_THREAD\_SUSPEND状态。

#### 11．控制线程

当需要对线程进行一些其他控制时，例如动态更改线程的优先级，可以调用如下函数

rt\_err\_t rt\_thread\_control(rt\_thread\_t thread, int cmd, void \*arg);

其中：thread为线程句柄；cmd指示控制命令；arg控制参数。

当控制执行正确时，返回RT\_EOK；当控制执行不正确时，返回RT\_ERROR。

指示控制命令cmd当前支持的命令包括：

（1）RT\_THREAD\_CTRL\_CHANGE\_PRIORITY：动态更改线程的优先级；

（2）RT\_THREAD\_CTRL\_STARTUP：开始运行一个线程，等同于rt\_thread\_startup()函数调用；

（3）RT\_THREAD\_CTRL\_CLOSE：关闭一个线程，等同于rt\_thread\_delete()或rt\_thread\_detach()函数调用。

#### 12．设置空闲挂钩

空闲挂钩函数是空闲线程的挂钩函数，如果设置了空闲挂钩函数，就可以在系统执行空闲线程时，自动执行空闲挂钩函数来做一些事情，如系统指示灯。设置空闲挂钩的函数如下：

rt\_err\_t rt\_thread\_idle\_sethook(void (\*hook)(void));

其中，hook为设置的挂钩函数。当设置成功时，返回RT\_EOK；当设置失败时，返回RT\_EFULL。

#### 13．删除空闲挂钩

删除空闲挂钩的函数如下：

rt\_err\_t rt\_thread\_idle\_delhook(void (\*hook)(void));

其中，hook为删除的挂钩函数。当删除成功时，返回RT\_EOK；当删除失败时，返回RT\_ENOSYS。

注：空闲线程是一个线程状态永远为就绪状态的线程，因此设置的挂钩函数必须保证空闲线程在任何时候都不会处于挂起状态，可能会导致线程挂起的函数都不能使用，比如rt\_thread\_delay()和rt\_sem\_take()。并且，由于malloc、free等存储器相关的函数内部使用了信号量作为临界区保护，因此在挂钩函数内部也不允许调用此类函数。

#### 14．设置调度器挂钩

在整个系统的运行过程中，系统都处于线程运行、中断触发-响应中断、切换到其他线程，甚至是线程间的切换过程中，或者说系统的上下文切换是系统中最普遍的事件。有时候用户想知道在一个时刻发生了什么样的线程切换，可以通过调用下面的函数设置一个相应的挂钩函数。在系统线程切换时，将调用该挂钩函数：

void rt\_scheduler\_sethook(void (\*hook)(rt\_thread\_t from, rt\_thread\_t to));

其中，hook表示用户定义的挂钩函数指针。

挂钩函数hook()声明如下：

void hook(struct rt\_thread\* from, struct rt\_thread\* to);

其中，from表示系统所要切换处的线程控制块指针；to表示系统要切换到的线程控制块指针。

注：仔细编写挂钩函数，稍有不慎将可能导致整个系统运行的不正常（在整个挂钩函数中，基本上不允许调用系统API，更不应该导致当前运行的上下文挂起）。

### 18.5.2 线程的创建及调度的实现

本节将在设计中添加设计代码，实现线程的创建及调度。

#### 打开设计工程

打开设计工程的主要步骤包括：

（1）启动Keil μVision（以下简称Keil）集成开发环境。

（2）在Keil主界面主菜单中，选择Project->Open Project...。

（3）弹出Select Project File对话框界面。在该界面中，将路径定位到下面的路径E:\STM32G0\_example\example\_18\_2\MDK-ARM。在该路径下，选中名字为top.uvprojx的工程文件。

（4）单击打开按钮，退出Select Project File对话框界面。

#### 2．添加UART控制台

在RT-Thread Nano上添加UART控制台打印功能后，就可以在代码中使用RT-Thread提供的打印函数rt\_kprintf()进行信息打印，从而获取自定义的打印信息，方便定义代码的缺陷或者获取系统当前运行状态等。实现控制台打印（需要确认rtconfig.h中已经使能RT\_USING\_CONSOLE宏定义），需要完成基本的硬件初始化，以及对接一个系统输出字符的函数。

1）初始化串口

在Keil主界面左侧的Project窗口中，找到并展开RTOS。在展开项中，双击board.c(kernel)，打开该设计文件。在该文件中，添加设计代码，如代码清单18-6所示。

代码清单18-6 添加串口初始化代码

static UART\_HandleTypeDef UartHandle;

static int uart\_init(void)

{

UartHandle.Instance = USART2;

UartHandle.Init.BaudRate = 115200;

UartHandle.Init.WordLength = UART\_WORDLENGTH\_8B;

UartHandle.Init.StopBits = UART\_STOPBITS\_1;

UartHandle.Init.Parity = UART\_PARITY\_NONE;

UartHandle.Init.Mode = UART\_MODE\_TX\_RX;

UartHandle.Init.HwFlowCtl = UART\_HWCONTROL\_NONE;

UartHandle.Init.OverSampling = UART\_OVERSAMPLING\_16;

if (HAL\_UART\_Init(&UartHandle) != HAL\_OK)

{

while (1);

}

return 0;

}

INIT\_BOARD\_EXPORT(uart\_init);

注：读者也可以在void rt\_hw\_board\_init()函数中，添加一行代码uart\_init();，来代替上面的INIT\_BOARD\_EXPORT(uart\_init);

2）添加rt\_hw\_console\_output函数

需要注意，RT-Thread系统中已有的打印均以\n结尾，而非\r\n，所以在字符输出时，需要在输出\n之前输出\r，完成回车与换行，否则系统打印出来的信息将只有换行。

在board.c文件中，添加的rt\_hw\_console\_output函数代码，如代码清单18-7所示。

代码清单18-7 rt\_hw\_console\_output函数代码

void rt\_hw\_console\_output(const char \*str)

{

rt\_size\_t i = 0, size = 0;

char a = '\r';

\_\_HAL\_UNLOCK(&UartHandle);

size = rt\_strlen(str);

for (i = 0; i < size; i++)

{

if (\*(str + i) == '\n')

{

HAL\_UART\_Transmit(&UartHandle, (uint8\_t \*)&a, 1, 1);

}

HAL\_UART\_Transmit(&UartHandle, (uint8\_t \*)(str + i), 1, 1);

}

}

3）保存board.c文件。

#### 3．添加创建和调度线程文件

本节将在设计中添加新的设计文件，在该文件中实现创建进程和线程调度的功能，主要步骤包括：

（1）在Keil主界面左侧窗口中，找到并选中Application/User文件夹，单击鼠标右键，出现浮动菜单。在浮动菜单内，选择Add New Item to Group ‘Application/User’...。

（2）弹出Add New Item to Group ‘Application/User’对话框界面。在该界面中，选择文件类型C File（.c），在Name：后面的文本框中输入thread\_sample，表示新创建的文件名字为thread\_sample.c。

（3）单击Add按钮，退出该对话框界面，同时自动打开thread\_sample.c文件。在该文件中，添加设计代码，如代码清单18-8所示。

代码清单18-8 thread\_sample.c文件

#include <rtthread.h>

#define THREAD\_PRIORITY 25

#define THREAD\_STACK\_SIZE 512

#define THREAD\_TIMESLICE 5

static rt\_thread\_t tid1 = RT\_NULL;

/\* 线程1的入口函数 \*/

static void thread1\_entry(void \*parameter)

{

unsigned char counter0=0;

while (1)

{

/\* 线程1采用低优先级运行，一直打印计数值 \*/

rt\_kprintf("thread1 count: %d\n", counter0++);

rt\_thread\_mdelay(500);

}

}

ALIGN(RT\_ALIGN\_SIZE)

static char thread2\_stack[1024];

static struct rt\_thread thread2;

/\* 线程2入口 \*/

static void thread2\_entry(void \*param)

{

unsigned char counter1=0;

while (1)

{

/\* 线程2采用低优先级运行，一直打印计数值 \*/

rt\_kprintf("thread2 count: %d\n", counter1++);

rt\_thread\_mdelay(500);

}

}

ALIGN(RT\_ALIGN\_SIZE)

static char thread3\_stack[1024];

static struct rt\_thread thread3;

/\* 线程3入口 \*/

static void thread3\_entry(void \*param)

{

unsigned char counter2=0;

for(counter2=0;counter2<20;counter2++)

{

/\* 线程3采用高优先级运行\*/

rt\_kprintf("thread3 count: %d\n", counter2);

}

rt\_kprintf("thread3 exit\n");

} /\* 线程3运行结束后也将自动脱离系统 \*/

int thread\_sample(void)

{

/\* 创建线程1，名称是thread1，入口是thread1\_entry\*/

tid1 = rt\_thread\_create("thread1",

thread1\_entry,

RT\_NULL,

THREAD\_STACK\_SIZE,

THREAD\_PRIORITY,

THREAD\_TIMESLICE);

/\* 如果获得线程控制块，启动这个线程 \*/

if (tid1 != RT\_NULL)

rt\_thread\_startup(tid1);

/\* 初始化线程2，名称是thread2，入口是thread2\_entry \*/

rt\_thread\_init(&thread2,

"thread2",

thread2\_entry,

RT\_NULL,

&thread2\_stack[0],

sizeof(thread2\_stack),

THREAD\_PRIORITY-1,

THREAD\_TIMESLICE);

rt\_thread\_startup(&thread2);

/\* 初始化线程3，名称是thread3，入口是thread3\_entry \*/

rt\_thread\_init(&thread3,

"thread3",

thread3\_entry,

RT\_NULL,

&thread3\_stack[0],

sizeof(thread3\_stack),

THREAD\_PRIORITY - 2,

THREAD\_TIMESLICE);

rt\_thread\_startup(&thread3);

return 0;

}

/\* 导出到 msh 命令列表中 \*/

MSH\_CMD\_EXPORT(thread\_sample, thread sample);

（4）保存该设计文件。

思考与练习18-1：说明RT-Thread中创建线程的方法。

思考与练习18-2：说明线程切换的方法，尤其是rt\_thread\_mdelay(500);代码（参考18.5.1节中切换线程的方法）。

#### 4．修改主函数

本节将修改主函数，主要步骤包括：

（1）在Keil主界面Project窗口中，找到并双击main.c，打开该设计文件。

（2）在main()主函数中，添加下面一行设计代码

thread\_sample();

（3）保存main.c文件。

#### 5．设计处理和测试

本节将对设计进行处理，并对该设计进行验证，主要包括：

（1）在Keil主界面主菜单中，选择Project->Build Target...，对该设计进行编译和连接，并生成可以下载到STM32G071 MCU内Flash存储器的文件格式。

（2）通过USB电缆，将NUCLEO-G071RB开发板的USB接口与PC/笔记本电脑的USB接口进行连接。

注：读者需要确认在电脑上虚拟出来的串口的端口号，对于Windows操作系统，读者可以在设备管理器中，找到虚拟的端口号，如图18.11所示，在此处虚拟出来的串口号为COM4。



图18.11 虚拟的串口端口号

（3）登陆下面的网址，https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html，读者根据自己电脑所使用的处理器和所安装的操作系统，下载PuTTY工具。在该设计中，使用的是64-bit x86 putty.exe文件，并将该文件下载到电脑桌面上。

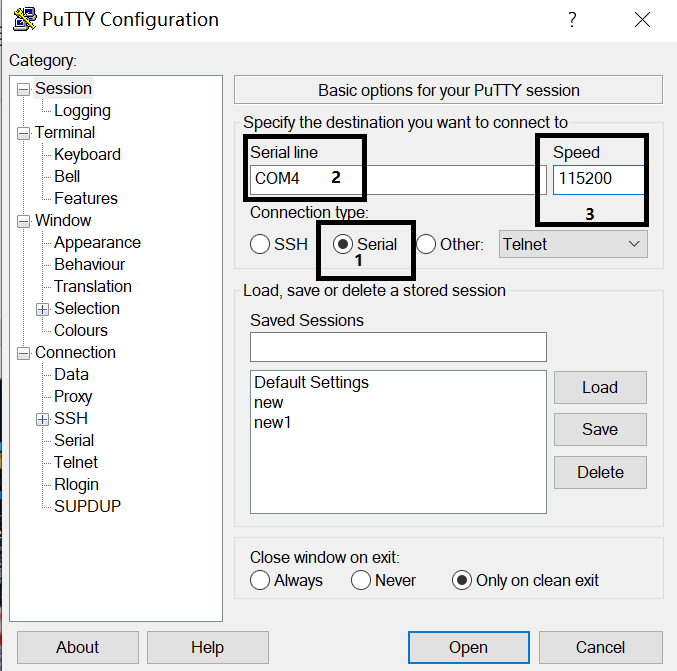


图18.11 PuTTY配置界面

（4）打开PuTTY工具，如图18.11所示，按图中标注的1、2和3的顺序，先选中Serial前面的复选框、在Serial line下面的文本框中输入图18.11中虚拟出来的串口号、在Speed下面的文本框中输入115200（该设置必须与工程中的串口初始化代码中所设置的波特率一致）。

（5）单击图18.11页面最下面的Open按钮，弹出一个黑色窗口。

（6）在Keil主界面主菜单中，选择Flash->Download。

（7）在COM4-PuTTY界面中，观察打印的信息，如图18.12所示。

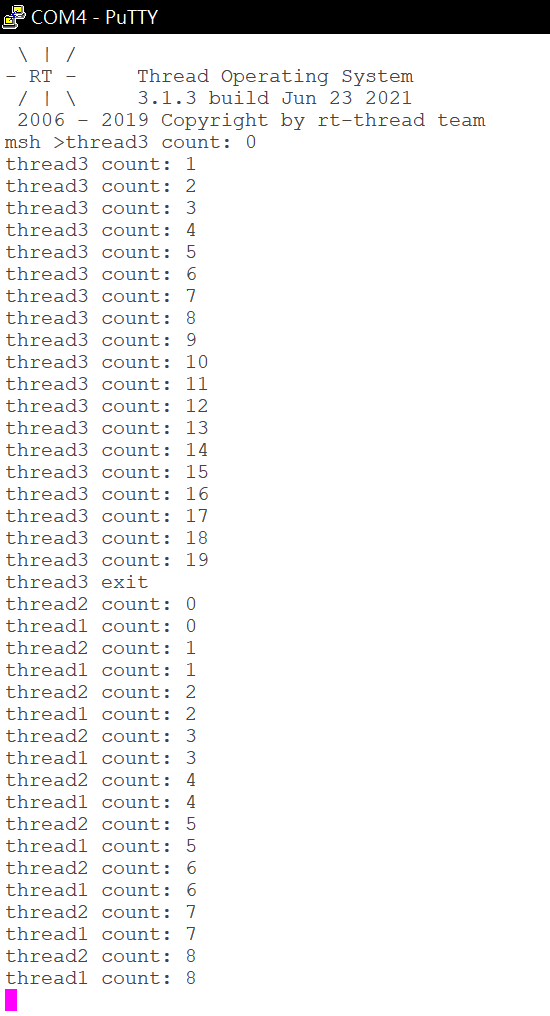


图18.12 在COM4-PuTTY打印消息（反色显示）

思考与练习18-3：根据打印信息，分析线程调度的过程和调度优先级，理解和掌握在RT-Thread创建线程的方法，以及最基本的线程调度方法。

### 18.5.3 定时器的使用

RT-Thread定时器由操作系统提供的一类接口（函数），它构建在芯片的硬件定时器基础之上，使系统能够提供不受数量限制的定时器服务。

前面提到，RT-Thread定时器分为HARD-TIMER与SOFT-TIMER，可以设置为单次定时与周期定时，这些属性均可在创建/初始化定时器时进行设置。如果没有设置HARD-TIMER或SOFT-TIMER，默认使用HARD\_TIMER。

#### 1．打开设计工程

打开设计工程的主要步骤包括：

（1）启动Keil μVision（以下简称Keil）集成开发环境。

（2）在Keil主界面主菜单中，选择Project->Open Project...。

（3）弹出Select Project File对话框界面。在该界面中，将路径定位到下面的路径E:\STM32G0\_example\example\_18\_3\MDK-ARM。在该路径下，选中名字为top.uvprojx的工程文件。

（4）单击打开按钮，退出Select Project File对话框界面。

#### 2．添加创建和调度定时器的文件

本节将在设计中添加新的设计文件，在该文件中实现创建定时器线程和调用定时器超时的功能，主要步骤包括：

（1）在Keil主界面左侧窗口中，找到并选中Application/User文件夹，单击鼠标右键，出现浮动菜单。在浮动菜单内，选择Add New Item to Group ‘Application/User’...。

（2）弹出Add New Item to Group ‘Application/User’对话框界面。在该界面中，选择文件类型C File（.c），在Name：后面的文本框中输入timer\_sample，表示新创建的文件名字为timer\_sample.c。

（3）单击Add按钮，退出该对话框界面，同时自动打开timer\_sample.c文件。在该文件中，添加设计代码，如代码清单18-9所示。

代码清单18-9 timer\_sample.c文件

#include <rtthread.h>

/\* 定时器的控制块 \*/

static rt\_timer\_t timer1;

static rt\_timer\_t timer2;

static int cnt = 0;

/\* 定时器1超时函数 \*/

static void timeout1(void \*parameter)

{

rt\_kprintf("periodic timer is timeout %d\n", cnt);

/\* 运行第10次，停止周期定时器 \*/

if (cnt++ >= 9)

{

rt\_timer\_stop(timer1);

rt\_kprintf("periodic timer was stopped! \n");

}

}

/\* 定时器2超时函数 \*/

static void timeout2(void \*parameter)

{

rt\_kprintf("one shot timer is timeout\n");

}

int timer\_sample(void)

{

/\* 创建定时器1 周期定时器 \*/

timer1 = rt\_timer\_create("timer1",

timeout1,

RT\_NULL,

10,

RT\_TIMER\_FLAG\_PERIODIC);

/\* 启动定时器1 \*/

if (timer1 != RT\_NULL) rt\_timer\_start(timer1);

/\* 创建定时器2 单次定时器 \*/

timer2 = rt\_timer\_create("timer2",

timeout2,

RT\_NULL,

30,

RT\_TIMER\_FLAG\_ONE\_SHOT);

/\* 启动定时器2 \*/

if (timer2 != RT\_NULL) rt\_timer\_start(timer2);

return 0;

}

/\* 导出到 msh 命令列表中 \*/

MSH\_CMD\_EXPORT(timer\_sample, timer sample);

（4）保存timer\_sample.c文件。

#### 3．修改主函数

本节将修改主函数，主要步骤包括：

（1）在Keil主界面Project窗口中，找到并双击main.c，打开该设计文件。

（2）在main()主函数中，添加下面一行设计代码

timer\_sample();

（3）保存main.c文件。

#### 4．设计处理和测试

本节将对设计进行处理，并对该设计进行验证，主要包括：

（1）在Keil主界面主菜单中，选择Project->Build Target...，对该设计进行编译和连接，并生成可以下载到STM32G071 MCU内Flash存储器的文件格式。

（2）通过USB电缆，将NUCLEO-G071RB开发板的USB接口与PC/笔记本电脑的USB接口进行连接。

（3）打开PuTTY工具，按18.5.2一节介绍的方法配置PuTTY内的串口通信参数。

（4）单击Open按钮，弹出一个黑色窗口。

（6）在Keil主界面主菜单中，选择Flash->Download。

（7）在COM4-PuTTY界面中，观察打印的信息，如图18.13所示。

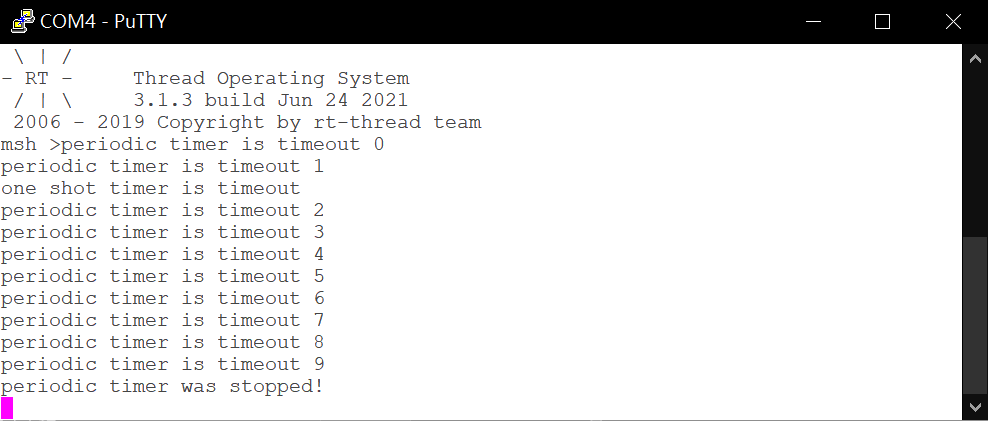


图18.13 在COM4-PuTTY打印消息

思考与练习18-4：根据图18.13给出的打印信息和设计代码，分析定时器所实现的功能。

### 18.5.4 互斥量的使用

互斥量是一种特殊的二值信号量。它和信号量不一样的是：拥有互斥量的线程拥有互斥量的所有权，互斥量支持递归访问且能防止线程优先级翻转；并且互斥量只能由持有线程释放，而信号量则可以由任何线程释放。

互斥量的使用比较单一，因为它是信号量的一种，并且它是以锁的形式存在。在初始化的时候，互斥量永远都处于开锁状态，而线程持有的时候则立刻转为闭锁的状态。

注：互斥量绝不能用在中断服务程序中。

#### 1．打开设计工程

打开设计工程的主要步骤包括：

（1）启动Keil μVision（以下简称Keil）集成开发环境。

（2）在Keil主界面主菜单中，选择Project->Open Project...。

（3）弹出Select Project File对话框界面。在该界面中，将路径定位到下面的路径E:\STM32G0\_example\example\_18\_4\MDK-ARM。在该路径下，选中名字为top.uvprojx的工程文件。

（4）单击打开按钮，退出Select Project File对话框界面。

#### 2．添加创建和使用互斥锁的文件

本节将在设计中添加新的设计文件，在该文件中实现创建定时器线程和调用定时器超时的功能，主要步骤包括：

（1）在Keil主界面左侧窗口中，找到并选中Application/User文件夹，单击鼠标右键，出现浮动菜单。在浮动菜单内，选择Add New Item to Group ‘Application/User’...。

（2）弹出Add New Item to Group ‘Application/User’对话框界面。在该界面中，选择文件类型C File（.c），在Name：后面的文本框中输入mutex\_sample，表示新创建的文件名字为mutex\_sample.c。

（3）单击Add按钮，退出该对话框界面，同时自动打开mutex\_sample.c文件。在该文件中，添加设计代码，如代码清单18-10所示。

代码清单18-9 mutex\_sample.c文件

#include <rtthread.h>

#define THREAD\_PRIORITY 8

#define THREAD\_TIMESLICE 5

/\* 指向互斥量的指针 \*/

static rt\_mutex\_t dynamic\_mutex = RT\_NULL;

static rt\_uint8\_t number1,number2 = 0;

ALIGN(RT\_ALIGN\_SIZE)

static char thread1\_stack[1024];

static struct rt\_thread thread1;

static void rt\_thread\_entry1(void \*parameter)

{

while(1)

{

//线程1获取到互斥量后，先后对number1、number2进行加1操作，然后

//释放互斥量

rt\_mutex\_take(dynamic\_mutex, RT\_WAITING\_FOREVER);

number1++;

rt\_thread\_mdelay(10);

number2++;

rt\_mutex\_release(dynamic\_mutex);

}

}

ALIGN(RT\_ALIGN\_SIZE)

static char thread2\_stack[1024];

static struct rt\_thread thread2;

static void rt\_thread\_entry2(void \*parameter)

{

while(1)

{

//线程2获取到互斥量后，检查number1、number2的值是否相同，

//相同则表示mutex起到了锁的作用

rt\_mutex\_take(dynamic\_mutex, RT\_WAITING\_FOREVER);

if(number1 != number2)

{

rt\_kprintf("not protect.number1 = %d, mumber2 = %d \n", number1 ,

number2);

}

else

{

rt\_kprintf("mutex protect ,number1 = mumber2 is %d\n",number1);

}

number1++;

number2++;

rt\_mutex\_release(dynamic\_mutex);

if(number1 >=50)

return;

}

}

/\* 互斥量示例的初始化 \*/

int mutex\_sample(void)

{

/\* 创建一个动态互斥量 \*/

dynamic\_mutex = rt\_mutex\_create("dmutex", RT\_IPC\_FLAG\_PRIO);

if (dynamic\_mutex == RT\_NULL)

{

rt\_kprintf("create dynamic mutex failed.\n");

return -1;

}

rt\_thread\_init(&thread1,

"thread1",

rt\_thread\_entry1,

RT\_NULL,

&thread1\_stack[0],

sizeof(thread1\_stack),

THREAD\_PRIORITY,

THREAD\_TIMESLICE);

rt\_thread\_startup(&thread1);

rt\_thread\_init(&thread2,

"thread2",

rt\_thread\_entry2,

RT\_NULL,

&thread2\_stack[0],

sizeof(thread2\_stack),

THREAD\_PRIORITY-1,

THREAD\_TIMESLICE);

rt\_thread\_startup(&thread2);

return 0;

}

/\* 导出到 msh 命令列表中 \*/

MSH\_CMD\_EXPORT(mutex\_sample, mutex sample);

#### 3．修改主函数

本节将修改主函数，主要步骤包括：

（1）在Keil主界面Project窗口中，找到并双击main.c，打开该设计文件。

（2）在main()主函数中，添加下面一行设计代码

mutex\_sample();

（3）保存main.c文件。

#### 4．设计处理和测试

本节将对设计进行处理，并对该设计进行验证，主要包括：

（1）在Keil主界面主菜单中，选择Project->Build Target...，对该设计进行编译和连接，并生成可以下载到STM32G071 MCU内Flash存储器的文件格式。

（2）通过USB电缆，将NUCLEO-G071RB开发板的USB接口与PC/笔记本电脑的USB接口进行连接。

（3）打开PuTTY工具，按18.5.2一节介绍的方法配置PuTTY内的串口通信参数。

（4）单击Open按钮，弹出一个黑色窗口。

（6）在Keil主界面主菜单中，选择Flash->Download。

（7）在COM4-PuTTY界面中，观察打印的信息，如图18.14所示。

思考与练习18-5：根据图18.14给出的打印信息和设计代码，分析互斥量在进程同步之间的作用，以及该设计使用互斥量所实现的功能。

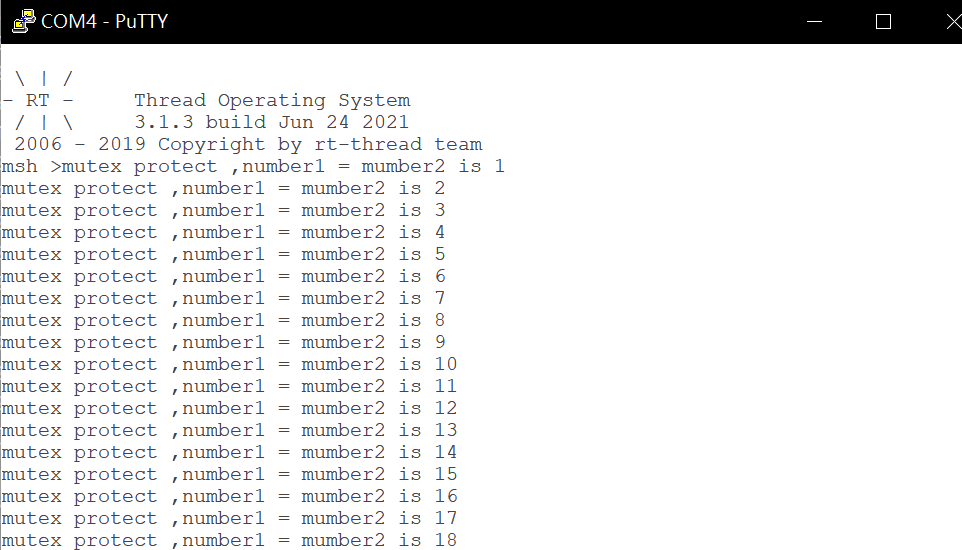


图18.14 在COM4-PuTTY打印消息

思考与练习18-3：根据打印信息，分析线程调度的过程和调度优先级，理解和掌握在RT-Thread创建线程的方法，以及最基本的线程调度方法。