初识RT-Thread

RT-Thread文档中心

2018-05-18

目录

目录		2
1	系统启动代码....................................	5
2	用户入口代码....................................	9
3	跑马灯的例子....................................	9
4	生产者-消费者问题...................................	11
5	其他例子	14

本文附带的例子是一个压缩包文件,将它解压,我们这里解压到本地。解压完成后的目 录结构如下图所示:

🔜 🛃 📑 🖛 ID:\im>	krt1052-evk		– 🗆 X
文件 主页 共享			~ 🕐
← → * ↑	此电脑 > work (D:) > imxrt1052-evk	✓ ⑦ 搜索"im:	rt1052-evk" ۶
> 📌 快速访问	名称 ^	修改日期	大小
	applications	2018/5/15 星期 文件夹	
> 🔜 此电脑	drivers	2018/5/15 星期 文件夹	
> 鹶 网络	Libraries	2018/5/15 星期 文件夹	
	🔒 packages	2018/5/15 星期 文件夹	
> •4 家庭组	rt-thread	2018/5/15 星期 文件夹	
	🚽 xip	2018/5/15 星期 文件夹	
	P .config	2018/5/15 星期 XML 配置文件	9 KB
	🗋 Kconfig	2018/5/15 星期 文件	14 KB
	🗋 project.ewd	2018/5/7 星期一 EWD 文件	100 KB
	🗋 project.ewp	2018/5/7 星期一 EWP 文件	66 KB
	🔮 project.eww	2018/5/7 星期— IAR IDE Works	р 1 КВ
	🗋 project.uvoptx	2018/5/15 星期 UVOPTX 文件	6 KB
	🔣 project.uvprojx	2018/5/15 星期 礦ision5 Projec	t 48 KB
	README.md	2018/5/11 星期 Markdown File	5 KB
	🗋 rtconfig.h	2018/5/15 星期 C/C++ 标头	5 KB
	rtconfig.py	2018/5/7 星期一 PY 文件	5 KB
	SConscript	2018/5/7 星期一 文件	1 KB
	SConstruct	2018/5/15 星期 文件	4 KB
	template.ewp	2018/5/7 星期一 EWP 文件	49 KB
	template.uvoptx	2018/5/7 星期一 UVOPTX 文件	6 KB
	🔣 template.uvprojx	2018/5/7 星期— 礦ision5 Projec	t 15 KB
21 个项目			

图 1: imxrt1052-evk代码目录

各个目录所包含的文件类型的描述如下表所示:

	描述
applications	
rt-thread	RT-Thread的源文件。
+components	RT-Thread的各个组件代码,例如finsh,lwip等。
+include	RT-Thread内核的头文件。
+libcpu	各类芯片的移植代码,此处包含了cortex-m7的移植文件。
+src	RT-Thread内核的源文件。
+tools	RT-Thread命令构建工具的脚本文件。
drivers	RT-Thread的驱动,不同平台的底层驱动具体实现。
Libraries	RT1052的固件库文件。
packages	RT-Thread的软件包,里面是一些内核例程。

在目录下,有一个project.uvprojx文件,它是本文内容所引述的例程中的一个MDK5工程文件,双击 "project.uvprojx"图标,打开此工程文件:

在工程主窗口的左侧"Project"栏里可以看到该工程的文件列表,这些文件被分别存放到如下几个组内,分别是:



图 2: 打开RT-Thread工程

目录组	描述
Applications	对应的目录为imxrt1052-evk/applications,它用于存放用户应用代 码-
Drivers	对应的目录为imxrt1052-evk/drivers,它用于存放RT-Thread底层的 驱动代码。
Libraries	对应的目录为imxrt1052-evk/Libraries/drivers,它用于存放RT1052 的固件库文件。
samples	对应的目录为imxrt1052-evk/drivers,它用于存放RT-Thread的内核 例程。
Kernel	对应的目录为imxrt1052-evk/src,它用于存放RT-Thread内核核心代 码。
CORTEX-M7	对应的目录为imxrt1052-evk/rt-thread/libcpu,它用于存放ARM Cortex-M7移植代码。
DeviceDrivers	对应的目录为imxrt1052-evk/rt-thread/components/drivers,它用于 存放RT-Thread驱动框架源码。
finsh	对应的目录为imxrt1052-evk/rt-thread/components/finsh,它用于存放RT-Thread命令行finsh命令行组件。
libc	对应的目录为imxrt1052-evk/rt-thread/components/libc,它用于存放RT-Thread使用的C库函数。

现在我们点击一下窗口上方工具栏中的按钮 🎬 ,对该工程进行编译,如图所示:

编译的结果显示在窗口下方的"Build"栏中,没什么意外的话,最后一行会显示"0 Error(s), * Warning(s).",即无任何错误。

在编译完 RT-Thread/imxrt1052-evk 后,我们可以通过 DAP 或 J-Link 下载到开发板上。

点击窗口上方的按钮 🌋 配置下载器

点击窗口上方的按钮 下载程序到开发板上

将开发板上 USB_232 端口通过数据线连接到电脑上,在电脑利用终端软件连接到此串口。

开发板复位,终端上显示下面的信息:

```
\ | /
- RT - Thread Operating System
/ | \ 3.0.4 build May 15 2018
2006 - 2018 Copyright by rt-thread team
using armcc, version: 5060750
build time: May 15 2018 20:58:41
msh >
```

我们可以通过输入Tab 输出当前系统所支持的所有命令。

1 系统启动代码

一般了解一份代码大多从启动部分开始,同样这里也采用这种方式,先寻找启动的源 头,因为MDK-ARM的用户程序入口为main()函数,所以先看看main()函数在哪个文件中。

躍 D:\rt-thread\bsp\imxrt1052-evk\project.uvprojx - μVision	_		\times		
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help					
	- 🗟 🕯	<u>۳</u> ه	- •		
Project 4 🖾 🗾 <u>main.c</u>			▼ X		
<pre>N3 Project project Applications Applica</pre>	am file m on or	ay be at	^ _		
E Pro & Fu 0, Te <			>		
Build Output			џ 🗴		
<pre>*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin' Build target 'RT-Thread IMXRT1052' After Build - User command #1: fromelfbin .\build\rtthread-imxrt.axfoutput rtthread-mdk.bin ".\build\rtthread-imxrt.axf" - 0 Error(s), 0 Warning(s). Build Time Elapsed: 00:00:01 </pre>					
🖅 Build Output 🙀 Find In Files					
		J-L	INK:		

图 3: 编译工程

Options for Target 'RT-Thread IMXRT1052'	×
Device Target Output Listing User C/C++	Asm Linker Debug Vtilities
Settings	
✓ Load Application at Startup ✓ Run to main() Initialization File:	I Load Application at Startup I Run to main() Initialization File:
Restore Debug Session Settings Breakpoints Toolbox Watch Windows & Performance Analyzer Memory Display System Viewer	Nubraries \arm\evkmimxrt 1050_flexspi_nc Edit Restore Debug Session Settings Image: Second Second Section Settings Image: Second Seco
CPU DLL: Parameter: SARMCM3.DLL -REMAP -MPU Dialog DLL: Parameter: DCM.DLL -pCM7	Driver DLL: Parameter: SARMCM3.DLL -MPU Dialog DLL: Parameter: TCM.DLL -pCM7
- Wam if outdated Executable is loaded Manage Component V	Wam if outdated Executable is loaded
OKCa	ncel Defaults Help

图 4: 下载器的配置界面

初识RT-Thread

1 系统启动代码



图 5: 下载成功界面

这里的main()函数位于Applications组的main.c文件中,它位于启动汇编代码后,与C代码的 入口跳转前。启动汇编在Libraries组的startup_MIMXRT1052.s中。

下面我们来看看main()函数中的这段代码:

```
//components.c中定义
/* re-define main function */
int $Sub$$main(void)
{
    rt_hw_interrupt_disable();
    rtthread_startup();
    return 0;
}
```

在这里\$Sub\$\$main函数仅仅调用了rtthread_startup()函数。RT-Thread支持多种平 台和多种编译器,而rtthread_startup()函数是RT-Thread规定的统一入口点,所以\$Sub \$\$main函数只需调用rtthread_startup()函数即可。例如采用GNU GCC编译器编译的RT-Thread,就是直接从汇编启动代码部分跳转到rtthread_startup()函数中,并开始第一 个C代码的执行的。在components.c的代码中找到rtthread_startup()函数,我们将可以看 到RT-Thread的启动流程:

```
int rtthread_startup(void)
{
    rt_hw_interrupt_disable();
    /* board level initalization
    * NOTE: please initialize heap inside board initialization.
    */
    rt_hw_board_init();
    /* show RT-Thread version */
    rt_show_version();
   /* timer system initialization */
    rt_system_timer_init();
    /* scheduler system initialization */
    rt_system_scheduler_init();
#ifdef RT_USING_SIGNALS
    /* signal system initialization */
    rt_system_signal_init();
#endif
    /* create init_thread */
    rt_application_init();
    /* timer thread initialization */
    rt_system_timer_thread_init();
    /* idle thread initialization */
```

```
rt_thread_idle_init();
/* start scheduler */
rt_system_scheduler_start();
/* never reach here */
return 0;
}
#endif
#endif
```

这部分启动代码,大致可以分为四个部分: -初始化与系统相关的硬件; -初始化系统 内核对象,例如定时器,调度器; -初始化系统设备,这个主要是为RT-Thread的设备框架 做的初始化; -初始化各个应用线程,并启动调度器。

2 用户入口代码

上面的启动代码基本上可以说都是和RT-Thread系统相关的,那么用户如何加入自己的 应用程序的初始化代码呢?RT-Thread将main函数作为了用户代码入口,只需要在main函 数里添加自己的代码即可。

```
int main(void)
{
   /* user app entry */
   return 0;
}
```

为了在进入main程序之前,完成系统功能初始化,可以使用*sub*和super函数标 识符在进入主程序之前调用另外一个例程,这样可以让用户不用去管main()之 前的系统初始化操作。详见ARM® Compiler v5.06 for µVision® armlink User Guide。

3 跑马灯的例子

对于从事电子方面开发的技术工程师来说,跑马灯大概是最简单的例子,就类似于每种 编程语言中程序员接触的第一个程序 Hello World 一样,所以这个例子就从跑马灯开始。创 建一个线程,让它定时地对LED进行更新(关或灭),例子对应的工程文件位于samples目 录下。

我们终端中输入 msh 命令: led_sample_init 然后回车就可以运行起来了,如图所示: 跑马灯例子

```
/*
 * 程序清单:跑马灯例程
 *
 * 跑马灯大概是最简单的例子,就类似于每种编程语言中程序员接触的第一个程序
 * Hello World一样,所以这个例子就从跑马灯开始。创建一个线程,让它定时地对
 * LED进行更新(关或灭)
 */
#include <rtthread.h>
```

初识RT-Thread

3 跑马灯的例子

₽	PuTTY	(inactiv	e)		_	×
msh	>led	samp	Le	init		~
msh	>led	lon, (col	int : 0		
led	off					
led	on,	count				
led	off					
led	on,	count		2		
led	off					
led	on,	count		3		
led	off					
led	on,	count		4		
led	off					
Led	on,	count	:	5		
Lea	OII					
led	on,	count		6		
lad	011	count				
led	off	count				
1ed	on.	count		8		
led	off					
led	on,	count		9		
led	off					
led	on,	count		10		
led	off					
led	on,	count		11		
led	off					
led	on,	count		12		
led	off					
led	on,	count	:	13		~

图 6: 运行跑马灯

```
#include <rtdevice.h>
ALIGN(RT_ALIGN_SIZE)
static rt_uint8_t led_stack[ 512 ];
/* 线程的TCB控制块 */
static struct rt_thread led_thread;
void rt_hw_led_init(void)
{
   rt_pin_mode(LED_PIN, PIN_MODE_OUTPUT);
}
static void led_thread_entry(void *parameter)
{
   unsigned int count = 0;
   rt_hw_led_init();
   while (1)
    {
       /* led1 on */
       rt_kprintf("led on, count : %d\r\n", count);
       count++;
       rt_pin_write(LED_PIN, 0);
       rt_thread_delay(RT_TICK_PER_SECOND / 2); /* sleep 0.5 second and switch to other thread */
       /* led1 off */
       rt_kprintf("led off\r\n");
```

```
rt_pin_write(LED_PIN, 1);
       rt_thread_delay(RT_TICK_PER_SECOND / 2);
   }
}
int led_sample_init(void)
{
   rt_err_t result;
   /* init led thread */
   result = rt_thread_init(&led_thread,
                           "led",
                           led_thread_entry,
                           RT NULL,
                           (rt_uint8_t *)&led_stack[0],
                           sizeof(led_stack),
                           20,
                           5);
   if (result == RT_EOK)
   {
       rt_thread_startup(&led_thread);
   }
   return 0;
}
   /* 如果设置了RT_SAMPLES_AUTORUN,则加入到初始化线程中自动运行 */
#if defined (RT SAMPLES AUTORUN) && defined(RT USING COMPONENTS INIT)
   INIT_APP_EXPORT(led_sample_init);
#endif
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(led_sample_init, led sample);
```

这里的rt_thread_delay(RT_TICK_PER_SECOND/2)函数的作用是延迟一段时间,即让 led线程休眠50个tick(按照rtconfig.h中的配置,1秒=RT_TICK_PER_SECOND个tick=100 tick,即在这份代码中延迟时间等于500ms)。在休眠的这段时间内,如果没有其他线程运 行,操作系统会切换到idle线程运行。

4 生产者-消费者问题

生产者-消费者问题是操作系统中的一个经典问题,在嵌入式操作系统中也经常能够遇 到,例如当串口接收到数据后,就将数据交给一个任务统一的进行处理。这里串口就如同一 个生产者产生数据,任务则如同一个消费者消费数据。在该例子中,我们将用RT-Thread的 编程模式编写一段代码来解决生产者-消费者问题。

我们在终端中输入msh命令: semaphore_producer_consumer_init 然后回车就可以运行起来了,如图所示:

```
/*
* 程序清单:生产者消费者例子
*
* 这个例子中将创建两个线程用于实现生产者消费者问题
*/
```

d	COM17 - PuTTY	_	\times	l
ms	h >semaphore producer consumer init		~	l
th	e producer generates a number: 1			l
th	e consumer[0] get a number: 1			l
ms	h >the producer generates a number: 2			l
th	e consumer[1] get a number: 2			l
th	e producer generates a number: 3			l
th	e consumer[2] get a number: 3			l
th	e producer generates a number: 4			l
th	e consumer[3] get a number: 4			l
th	e producer generates a number: 5			l
th	e consumer[4] get a number: 5			l
th	e producer generates a number: 6			l
th	e consumer[0] get a number: 6			l
th	e producer generates a number: 7			l
th	e consumer[1] get a number: 7			l
th	e producer generates a number: 8			l
th	e consumer[2] get a number: 8			l
th	e producer generates a number: 9			l
th	e consumer[3] get a number: 9			l
th	e producer generates a number: 10			l
th	e consumer[4] get a number: 10			l
th	e consumer[0] sum is 55			l
t	he consumer[1083179008] exit!			
th	e producer exit!			
			\sim	I

图 7: 运行生产者-消费者问题

```
#include <rtthread.h>
#define THREAD_PRIORITY
                        6
#define THREAD_STACK_SIZE
                          512
#define THREAD_TIMESLICE
                          5
/* 定义最大5个元素能够被产生 */
#define MAXSEM 5
/* 用于放置生产的整数数组 */
rt_uint32_t array[MAXSEM];
/* 指向生产者、消费者在array数组中的读写位置 */
static rt_uint32_t set, get;
/* 指向线程控制块的指针 */
static rt_thread_t producer_tid = RT_NULL;
static rt_thread_t consumer_tid = RT_NULL;
struct rt_semaphore sem_lock;
struct rt_semaphore sem_empty, sem_full;
/* 生成者线程入口 */
void producer_thread_entry(void *parameter)
{
   int cnt = 0;
   /* 运行10次 */
   while (cnt < 10)
   {
       /* 获取一个空位 */
       rt_sem_take(&sem_empty, RT_WAITING_FOREVER);
```

}

{

```
/* 修改array内容, 上锁 */
       rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
       array[set % MAXSEM] = cnt + 1;
       rt_kprintf("the producer generates a number: %d\n", array[set % MAXSEM]);
       set++;
       rt_sem_release(&sem_lock);
       /* 发布一个满位 */
       rt_sem_release(&sem_full);
       cnt++;
       /* 暂停一段时间 */
       rt_thread_delay(50);
   }
   rt_kprintf("the producer exit!\n");
/* 消费者线程入口 */
void consumer_thread_entry(void *parameter)
   rt_uint32_t no;
   rt_uint32_t sum;
   /* 第n个线程,由入口参数传进来 */
   no = (rt_uint32_t)parameter;
   sum = 0;
   while (1)
   {
       /* 获取一个满位 */
       rt_sem_take(&sem_full, RT_WAITING_FOREVER);
       /* 临界区, 上锁进行操作 */
       rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
       sum += array[get % MAXSEM];
       rt_kprintf("the consumer[%d] get a number: %d\n", (get % MAXSEM), array[get % MAXSEM]);
       qet++;
       rt_sem_release(&sem_lock);
       /* 释放一个空位 */
       rt_sem_release(&sem_empty);
       /* 生产者生产到10个数目,停止,消费者线程相应停止 */
       if (get == 10) break;
       /* 暂停一小会时间 */
       rt_thread_delay(10);
```

```
}
   rt_kprintf("the consumer[%d] sum is %d \n ", no, sum);
   rt_kprintf("the consumer[%d] exit!\n");
}
int semaphore_producer_consumer_init()
{
   /* 初始化3个信号量 */
   rt_sem_init(&sem_lock, "lock",
                                            RT_IPC_FLAG_FIF0);
                                   1,
   rt_sem_init(&sem_empty, "empty",
                                   MAXSEM, RT_IPC_FLAG_FIFO);
   rt_sem_init(&sem_full, "full",
                                    0,
                                           RT_IPC_FLAG_FIFO);
   /* 创建线程1 */
   producer_tid = rt_thread_create("producer",
                                  producer_thread_entry, RT_NULL, /* 线程入口是producer_thread_entry, 入口参
                                  THREAD_STACK_SIZE, THREAD_PRIORITY - 1, THREAD_TIMESLICE);
   if (producer_tid != RT_NULL)
       rt_thread_startup(producer_tid);
   /* 创建线程2 */
   consumer_tid = rt_thread_create("consumer",
                                  consumer_thread_entry, RT_NULL, /* 线程入口是consumer_thread_entry, 入口参
                                  THREAD_STACK_SIZE, THREAD_PRIORITY + 1, THREAD_TIMESLICE);
   if (consumer_tid != RT_NULL)
       rt_thread_startup(consumer_tid);
   return 0;
}
/* 如果设置了RT_SAMPLES_AUTORUN,则加入到初始化线程中自动运行 */
#if defined (RT_SAMPLES_AUTORUN) && defined(RT_USING_COMPONENTS_INIT)
   INIT_APP_EXPORT(semaphore_producer_consumer_init);
#endif
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(semaphore_producer_consumer_init, producer_consumer sample);
```

该例创建了两个线程,一个作为生产者,一个作为消费者。-生产者线程将cnt值每次 加1并循环存入array数组的5个成员内; - 消费者线程将生产者中生产的数值打印出来,并 累加求和。

5 其他例子

其他更多的内核例子可以从samples目录下找到,加入到工程中就能运行起来。