**在Linux下实现FreeRTOS的简单模拟器**

**FreeRTOS**

基础知识不赘述，请参考[朱工的专栏](http://blog.csdn.net/zhzht19861011/article/details/49819109" \o "朱工的专栏" \t "_blank)， 本文主要描述怎么在Linux的环境下跑一个FreeRTOS的模拟器

**官方示例**

FreeRTOS的官方提供了一个在[Linux下的Simulator的示例](https://www.freertos.org/FreeRTOS-simulator-for-Linux.html" \t "_blank)，但是用的Kernel的版本非常老，是V6的版本，FreeRTOS现在已经进化到V10了，作为一个标准码农，不用最新版本简直不舒服斯基 >\_<。   
先把官方示例下载下来，在官方示例中，有一个Debug和Release的目录，在这两个目录下使用make all命令就可以直接编出来可执行文件在Linux下直接运行，当然，使用Eclipse直接打开对应的工程来编译也是可以的。

**更新Kernel**

先创建一个文件夹Simulator\_Linux，其下有三个目录

FreeRTOS\_Kernel

inc

src

FreeRTOS\_Kernel中保存内核代码，inc和src保存APP的代码，当然，可以按照自己的爱好自行调整目录结构。

再去[FreeRTOS官网](https://www.freertos.org/a00104.html)下载最新的Kernel代码，解压后进入FreeRTOS\Source目录。   
按照官方的示例，将最新的代码拷贝到FreeRTOS\_Kernel目录中。   
include目录中的头文件不管三七二十一全拷贝过来即可（我懒，不想一个个去梳理>\_<）。   
.c文件只需要拷贝croutine.c, list.c, queue.c以及tasks.c即可。（croutine其实也可以不用拷贝，但是要做一些配置）   
portable文件夹不从FreeRTOS\Source拷贝，而从simulator的示例中拷贝（\Posix\_GCC\_Simulator\FreeRTOS\_Posix\FreeRTOS\_Kernel）

退回到上层目录，在将官方示例的simulator的根目录下的FreeRTOSConfig.h拷贝到inc目录下。   
在src目录下创建main.c文件，在其中定义一个空的main函数即可。

此时，我们就拥有了一个完整的Kernel的代码，当然这个时候还是没法编译的，一来缺少makefile，二来portable的文件与最新的Kernel其实并不完全匹配。当然，APP的代码也就是main函数的代码也还是空的。

**Makefile**

参考官方示例的makefile，在根目录下创建Makefile文件，同样在子目录下也包含两个subdir.mk用来编译需要的对应的.o，具体不再赘述：   
Makefile:

RM := rm -rf

PROJ\_ROOT :=.

BUILD\_TMP :=$(PROJ\_ROOT)/tmp

TARGET\_INC := -I$(PROJ\_ROOT)/inc \

-I$(PROJ\_ROOT)/FreeRTOS\_Kernel/include \

-I$(PROJ\_ROOT)/FreeRTOS\_Kernel/portable/GCC/Posix

-include subdir.mk

-include FreeRTOS\_Kernel/subdir.mk

ifneq ($(MAKECMDGOALS),clean)

ifneq ($(strip $(C\_DEPS)),)

-include $(C\_DEPS)

endif

endif

all:simulator\_linux.bin

simulator\_linux.bin: $(OBJS)

@echo 'Building target: $@'

gcc -pthread -lrt -o"simulator\_linux.bin" $(OBJS) $(LIBS)

@echo 'Finished building target: $@'

@echo ' '

clean:

-$(RM) $(OBJS)$(C\_DEPS)$(EXECUTABLES) simulator\_linux.bin

-@echo ' '

.PHONY: all clean dependents

.SECONDARY:

subdir.mk:

C\_SRCS += \

$(PROJ\_ROOT)/src/main.c

OBJS += \

$(BUILD\_TMP)/main.o

C\_DEPS += \

$(BUILD\_TMP)/main.d

# Each subdirectory must supply rules for building sources it contributes

$(BUILD\_TMP)/%.o: $(PROJ\_ROOT)/src/%.c

@echo 'Building file: $<'

gcc -DUSE\_STDIO=1 -D\_\_GCC\_POSIX\_\_=1 $(TARGET\_INC) -O2 -Wall -c -fmessage-length=0 -pthread -lrt -MMD -MP -MF"$(@:%.o=%.d)" -MT"$(@:%.o=%.d)" -o"$@" "$<"

@echo 'Finished building: $<'

@echo ' '

FreeRTOS\_Kernel/subdir.mk

C\_SRCS += \

$(PROJ\_ROOT)/FreeRTOS\_Kernel/croutine.c \

$(PROJ\_ROOT)/FreeRTOS\_Kernel/list.c \

$(PROJ\_ROOT)/FreeRTOS\_Kernel/queue.c \

$(PROJ\_ROOT)/FreeRTOS\_Kernel/tasks.c \

$(PROJ\_ROOT)/FreeRTOS\_Kernel/portable/GCC/Posix/port.c \

$(PROJ\_ROOT)/FreeRTOS\_Kernel/portable/MemMang/heap\_3.c

OBJS += \

$(BUILD\_TMP)/croutine.o \

$(BUILD\_TMP)/list.o \

$(BUILD\_TMP)/queue.o \

$(BUILD\_TMP)/tasks.o \

$(BUILD\_TMP)/port.o \

$(BUILD\_TMP)/heap\_3.o

C\_DEPS += \

$(BUILD\_TMP)/croutine.d \

$(BUILD\_TMP)/list.d \

$(BUILD\_TMP)/queue.d \

$(BUILD\_TMP)/tasks.d \

$(BUILD\_TMP)/port.d \

$(BUILD\_TMP)/heap\_3.d

$(BUILD\_TMP)/%.o: $(PROJ\_ROOT)/FreeRTOS\_Kernel/%.c

@echo 'Building file: $<'

gcc -DUSE\_STDIO=1 -D\_\_GCC\_POSIX\_\_=1 $(TARGET\_INC) -O2 -Wall -c -fmessage-length=0 -pthread -lrt -MMD -MP -MF"$(@:%.o=%.d)" -MT"$(@:%.o=%.d)" -o"$@" "$<"

@echo 'Finished building: $<'

@echo ' '

$(BUILD\_TMP)/%.o: $(PROJ\_ROOT)/FreeRTOS\_Kernel/portable/GCC/Posix/%.c

@echo 'Building file: $<'

gcc -DUSE\_STDIO=1 -D\_\_GCC\_POSIX\_\_=1 $(TARGET\_INC) -O2 -Wall -c -fmessage-length=0 -pthread -lrt -MMD -MP -MF"$(@:%.o=%.d)" -MT"$(@:%.o=%.d)" -o"$@" "$<"

@echo 'Finished building: $<'

@echo ' '

$(BUILD\_TMP)/%.o: $(PROJ\_ROOT)/FreeRTOS\_Kernel/portable/MemMang/%.c

@echo 'Building file: $<'

gcc -DUSE\_STDIO=1 -D\_\_GCC\_POSIX\_\_=1 $(TARGET\_INC) -O2 -Wall -c -fmessage-length=0 -pthread -lrt -MMD -MP -MF"$(@:%.o=%.d)" -MT"$(@:%.o=%.d)" -o"$@" "$<"

@echo 'Finished building: $<'

@echo ' '

OK，编译体系已经搞好。此时在根目录下直接敲 make all 就应该可以进行编译啦。

**配置更新**

此时直接make all会发现有一大堆错误，这是因为FreeRTOS的版本更新后一些结构体的名字发生了变化，在FreeRTOS.h中有一个兼容性的宏可以控制一部分的兼容性，但是因为版本跨度比较大， 我们依然需要在portmacro.h中做适当的适配：

/\*-----------------------------------------------------------\*/

typedef portSTACK\_TYPE StackType\_t;

typedef portBASE\_TYPE BaseType\_t;

typedef unsigned long UBaseType\_t;

#define portTICK\_PERIOD\_MS ( ( TickType\_t ) 1000 / configTICK\_RATE\_HZ )

#if( configUSE\_16\_BIT\_TICKS == 1 )

typedef unsigned portSHORT TickType\_t;

#define portMAX\_DELAY ( TickType\_t ) 0xffff

#else

typedef unsigned portLONG TickType\_t;

#define portMAX\_DELAY ( TickType\_t ) 0xffffffff

#endif

/\*-----------------------------------------------------------\*/

/\*

#if( configUSE\_16\_BIT\_TICKS == 1 )

typedef unsigned portSHORT portTickType;

#define portMAX\_DELAY ( portTickType ) 0xffff

#else

typedef unsigned portLONG portTickType;

#define portMAX\_DELAY ( portTickType ) 0xffffffff

#endif

\*/

/\*-----------------------------------------------------------\*/

**编译运行**

此时再make clean后重新make all，编译即可通过。但是实际上main.c里并没有执行任何代码，所以感受不到FreeRTOS的实际效果，我们在main.c中添加一些代码，来创建两个任务并通过消息队列来传递一些数据：

#include <stdio.h>

#include <stdlib.h>

#include "main.h"

#include "FreeRTOS.h"

#include "task.h"

#include "queue.h"

static void vTask1( void \*pvParameters );

static void vTask2( void \*pvParameters );

int main()

{

static xQueueHandle xTestQueue;

xTestQueue = xQueueCreate( 10, ( unsigned portBASE\_TYPE ) sizeof( unsigned short ) );

xTaskCreate( vTask1, "vTask1", configMINIMAL\_STACK\_SIZE, ( void \* ) &xTestQueue, tskIDLE\_PRIORITY, NULL );

xTaskCreate( vTask2, "vTask2", configMINIMAL\_STACK\_SIZE, ( void \* ) &xTestQueue, tskIDLE\_PRIORITY, NULL );

vTaskStartScheduler();

return 1;

}

static void vTask1( void \*pvParameters )

{

unsigned short usValue = 0, usLoop;

xQueueHandle \*pxQueue;

const unsigned short usNumToProduce = 3;

short sError = pdFALSE;

pxQueue = ( xQueueHandle \* ) pvParameters;

for( ;; )

{

for( usLoop = 0; usLoop < usNumToProduce; ++usLoop )

{

/\* Send an incrementing number on the queue without blocking. \*/

printf("Task1 will send: %d\r\n", usValue);

if( xQueueSendToBack( \*pxQueue, ( void \* ) &usValue, ( portTickType ) 0 ) != pdPASS )

{

sError = pdTRUE;

}

else

{

++usValue;

}

}

vTaskDelay( 2000 );

}

}

static void vTask2( void \*pvParameters )

{

unsigned short usData = 0;

xQueueHandle \*pxQueue;

pxQueue = ( xQueueHandle \* ) pvParameters;

for( ;; )

{

while( uxQueueMessagesWaiting( \*pxQueue ) )

{

if( xQueueReceive( \*pxQueue, &usData, ( portTickType ) 0 ) == pdPASS )

{

printf("Task2 received:%d\r\n", usData);

}

}

vTaskDelay( 5000 );

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* This is a stub function for FreeRTOS\_Kernel \*/

void vMainQueueSendPassed( void )

{

return;

}

/\* This is a stub function for FreeRTOS\_Kernel \*/

void vApplicationIdleHook( void )

{

return;

}

再次重新编译，执行编译后在根目录下生成的simulator\_linux.bin，即可看到两个Task之间的交互过程：   


**总结**

FreeRTOS的内核极其小巧，只需要几个简单的文件就可以进行编译运行，当在不同的硬件上进行移植的时候，只需要修改portable目录里的文件即可完成对硬件的适配，实际上官方也提供了大量的已经完成移植的设备的portable文件，我们只需要简单的拷贝过来即可