# Embedded electronics exam: running FreeRTOS on the Red Pitaya running GNU Linux

J.-M Friedt, December 28, 2024

FreeRTOS provides a demonstration of its execution on top of a POSIX compliant operating system. We aim at showing how FreeRTOS can be executed on the Red Pitaya running GNU Linux.

1. Demonstrate how you compile the demonstration of FreeRTOS running on a POSIX compliant environment by downloading the latest FreeRTOS archive and using the files in `FreeRTOS/Demo/Posix_GCC`. We shall be interested in the simplest demonstration `main_blinky.c`: update the C files needed to compile this demonstation according to the instructions provided in the header of these files. Execute on the x86 host computer running GNU Linux to demonstrate the functional result. When executed, the demonstation must start with

   ```
   Trace started.
   The trace will be dumped to disk if a call to configASSERT() fails.
   Starting echo blinky demo
   Message received from task
   Message received from task
   Message received from task
   ...
   ```

   and *not* with

   ```
   The trace will be dumped to disk if a call to configASSERT() fails.
   Starting full demo
   ```

   which would mean failure to properly configure FreeRTOS for the demonstration.

2. Modify the `main_blinky.c` example to launch three tasks, one displaying a "Hello World" message every second, one displaying alternatively "LED1 0" and "LED1 1" every 500 ms, and one displaying alternatively "LED2 0" and "LED2 1" every 300 ms. Execute on the x86 host computer running GNU Linux to demonstrate the functional result.

3. Since the Red Pitaya is also running GNU Linux, we wish to demonstrate the functional execution of FreeRTOS on this embedded platform. Cross-compile the FreeRTOS executable towards the ARM CPU, and execute on the Red Pitaya. Remember that the Red Pitaya Buildroot repository is found in `/home/jmfriedt/buildroot-2023.08-rc3_redpit` of the university computers, where you can find the cross-compiler at the appropriate location. How do you check (which command?) that the resulting executable is compiled for the appropriate target architecture?

4. Since the Red Pitaya is fitted with two CPU-accessible LEDs, we wish to replace the "ON1" and "OFF1" with blinking one of the LEDs, and "ON2" and "OFF2" with blinking the other LED. Be aware that writing to a pseudo-file in `/sys` requires seeking the beginning of the file every time a new command is written see the `fseek()` instruction). Demonstrate how you modify the FreeRTOS program to achieve this result by executing the updated software on the Red Pitaya.

5. Rather than having two separate functions blinking the two LEDs with different time intervals, we wish to have a single function launched twice by the scheduler, this unique function receiving two arguments when launched, namely the index of the LED and the delay between two state transitions. Demonstrate how you modify the function called when creating the tasks so that these two arguments are provided and interpreted properly.

6. Which aspect(s) of the POSIX compliance of the underlying operating system made ~~this last~~ step 4 much easier than if we had worked at the baremetal C programming level?

7. Comment on executing FreeRTOS on top of the POSIX compliant operating system as opposed to nativey executing the executive environment monolithic application: what are the pros and cons of each approach? What additional work would be involved for executing FreeRTOS as an independent application on the Red Pitaya not running GNU/Linux?

## Answers:

1. As stated in the header of the C files, "If mainSELECTED_APPLICATION = BLINKY_DEMO the simple blinky demo will be built. The simply blinky demo is implemented and described in main_blinky.c." so we add at the beginning of `main.c`:
   `#define mainSELECTED_APPLICATION BLINKY_DEMO`
   compile with `make` and execute `build/posix_demo`

2. the default example is rather complex and involves Queues, so in `main_blinky.c` we remove the provided tasks and replace with

```
static void vTask0( void * pvParameters )
{while (1)
   {printf("Hello World\n");
    vTaskDelay( 1000 / portTICK_PERIOD_MS );
   }
}

static void vTask1( void * pvParameters )
{int val=0;
 while (1)
       {printf("LED1 %d\n",val);
        val=1-val;
        vTaskDelay( 500 / portTICK_PERIOD_MS );
       }
}

static void vTask2( void * pvParameters )
{int val=0;
 while (1)
       {printf("LED2 %d\n",val);
        val=1-val;
        vTaskDelay( 300 / portTICK_PERIOD_MS );
       }
}
```

   whose prototype is declared before the main() function

```
static void vTask0( void * pvParameters );
static void vTask1( void * pvParameters );
static void vTask2( void * pvParameters );
```

   and scheduled using in the main() function

```
xTaskCreate( vTask0, "T0", configMINIMAL_STACK_SIZE, NULL, mainQUEUE_SEND_TASK_PRIORITY, NULL);
xTaskCreate( vTask1, "T1", configMINIMAL_STACK_SIZE, NULL, mainQUEUE_SEND_TASK_PRIORITY, NULL);
xTaskCreate( vTask2, "T2", configMINIMAL_STACK_SIZE, NULL, mainQUEUE_SEND_TASK_PRIORITY, NULL);
vTaskStartScheduler();
```

3. export the path to the buildroot toolchain and compile with `arm-linux-gcc`, replacing at the beginning of the Makefile

```
CC := gcc
```

   with

```
CC := arm-linux-gcc
```

The resulting executable

```
$ file build/posix_demo
build/posix_demo: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV)
```

is indeed ready to run on a 32-bit ARM CPU.

4. as an example of blinking the LEDs the basic program

```
#include <stdio.h>
#include <unistd.h> // sleep

int main()
{FILE *f;
 int val=0;
 f=fopen("/sys/class/leds/led8/brightness","w");
 while (1)
  {fseek(f,0, SEEK_SET);
   fprintf(f,"%d\n",val);
   val=1-val;
   sleep(1);
  }
}
```

does blink the LED after we have disabled the trigger from the shell: this template is used to replace the content of vTask1 and vTask2

```
redpitaya> cd /sys/class/leds/led8
redpitaya> echo "none" > trigger
redpitaya> cd /sys/class/leds/led9
redpitaya> echo "none" > trigger
```

Now the tasks are

```
static void vTask1( void * pvParameters )
{FILE *f;
 int val=0;
 f=fopen("/sys/class/leds/led8/brightness","w");
 while (1)
  {fseek(f,0, SEEK_SET);
   fprintf(f,"%d\n",val);
   val=1-val;
   vTaskDelay( 500 / portTICK_PERIOD_MS );
  }
}

static void vTask2( void * pvParameters )
{FILE *f;
 int val=0;
 f=fopen("/sys/class/leds/led9/brightness","w");
 while (1)
  {fseek(f,0, SEEK_SET);
   fprintf(f,"%d\n",val);
   val=1-val;
   vTaskDelay( 300 / portTICK_PERIOD_MS );
  }
}
```

while vTask0 remains unchanged.

5. we wish to provide two arguments, the delay and the LED index, so that a structure must be created.

```
struct arg {int led;int delay;};

void main_blinky( void )
{static struct arg arg1={.led=8,.delay=500},arg2={.led=9,.delay=300};
 xTaskCreate( vTask0,"T0",configMINIMAL_STACK_SIZE, NULL, mainQUEUE_SEND_TASK_PRIORITY, NULL);
 xTaskCreate( vTask1,"T1",configMINIMAL_STACK_SIZE, &arg1, mainQUEUE_SEND_TASK_PRIORITY, NULL);
 xTaskCreate( vTask1,"T2",configMINIMAL_STACK_SIZE, &arg2, mainQUEUE_SEND_TASK_PRIORITY, NULL);
 vTaskStartScheduler();

 for( ; ; )
    {
    }
}

static void vTask1( void * pvParameters )
{FILE *f;
 char s[64];
 struct arg *argument;
 argument=(struct arg*)pvParameters;
 int val=0;
 printf("%d %d\n",argument->led,argument->delay);
 sprintf(s,"/sys/class/leds/led%d/brightness",argument->led);
 f=fopen(s,"w");
 while (1)
  {fseek(f,0, SEEK_SET);
   fprintf(f,"%d\n",val);
   val=1-val;
   vTaskDelay( argument->delay / portTICK_PERIOD_MS );
  }
}
```

The arguments must be prefixed with `static` to be stored on the heap and not on the stack which would be corrupted once the scheduler is launched.

6. the POSIX compliance provides a filesystem (`/sys`) and drivers for accessing the hardware peripherals without the need to identify in the datasheet the registers for handling the GPIOs controlling the LEDs

7. running the baremetal FreeRTOS application would require initializing the stack, running from the SD card and hence implement a basic bootloader, and accessing the registers controlling the LEDs since FreeRTOS does not abstract hardware with drivers.