Digital electronics

J.-M Friedt

FEMTO-ST/time & frequency department

jmfriedt@femto-st.fr

slides at `jmfriedt.free.fr`

January 12, 2023

Beyond C ...

Monolithic programming in C: a single binary output includes all functions after linkage of all the source files converted to objects + libraries

⇒ specify function prototypes, argument exchange and returned, make sure that each function performs as expected (unit test)

Alternative:

- each programmer develops their own task
- a supervisor schedules these tasks
- **challenge:** multiples tasks, one resource: how to make make sure the state of the resource is known when accessed by multiple tasks?
- **challenge:** tasks wish to share data ⇒ how to make sure the shared data are in a known state?
- code portability: hardware abstraction (drivers) + provide homogeneous interfaces ("temperature sensors", "asynchronous communication port"...)

⇒ executive environment and operating systems (list of system calls providing a link between userspace and a supervisor – the kernel)

Plan

7 lessons/lab sessions (4-hour long schedules):

1. Executive environments: principles and introduction, getting started with FreeRTOS
2. FreeRTOS, RTEMS, Nuttx ... multitasking and associated methods to make sure shared data and resources are kept in known states (mutex & semaphore)
3. Using the scheduler, mutex and semaphores to solve the "philosopher problem"
4. Embedded systems with GNU/Linux – POSIX compatible operating system
   Architecture of an operating system, kernel v.s userspace
   Internet connectivity and networking
5. Accessing hardware resources from userspace – memory translation from physical to virtual address space (Memory Management Unit) – `/dev/mem`
6. Accessing hardware resources from a web server – internet connected instrument
7. From userspace to kernel space: character device (*char device*) for communicating between users and the kernel

Beyond C ... executive environments

- add an additional abstraction layer (assembly – C – kernel)
- executive environments: develop as if an operating system was available, but generate a single, monolithic binary application.
- Scheduler $\Rightarrow$ multiple tasks appear as if executed in parallel $\Rightarrow$ concept of priority.
- Allows for multiple programmer to develop in parallel, each providing one functional task.
- The executive environment provides the means for synchronizing tasks and associated data handling.
- Application portablility: hardware layers might be hidden from the developer (hardware abstraction layers), exhibiting standardized and platform independent interfaces.
- But an addition constraint: understand a new set of protocols and programming methods (API – Application Programming Interface)

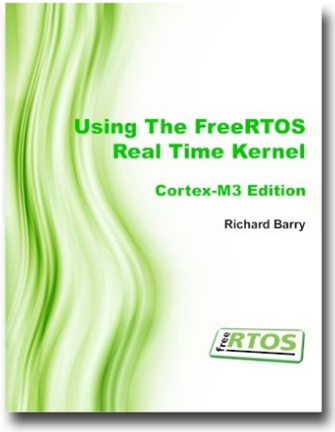
Executive environments

- **No** dynamic library or program loading and linking: static code
- concept of tasks ...
- ... and the assocated challenges (concurrent access to resources, to variables, prioritites),
- Solutions: mutex, semaphores.
- Appropriate for parallel task development and code re-usage
- Some free, opensource executive environments: select depending on available resources and supported platforms: FreeRTOS[1], TinyOS[2], RTEMS[3], NuttX[4], Zephyr Project [5], ChibiOS [6]
- the scheduler must never starve of processes to run
  $\Rightarrow$ FreeRTOS' `xTaskCreate()` schedules tasks in an infinite loop.

[1] `www.freertos.org`
[2] `www.tinyos.net` but no update since 2013
[3] Real-Time Executive for Multiprocessor Systems at `www.rtems.com`, used by ESA
[4] `nuttx.org` & `nuttx.apache.org`
[5] `www.zephyrproject.org`
[6] `www.chibios.org/` used for example in `github.com/ttrftech/NanoVNA`

FreeRTOS

Portable executive environment written by R. Barry [7] [8] (now bought by AWS) whose functionalitites are restricted to 6 files !



- **no** hardware abastraction/drivers
- no dynamically loaded executable or library
- scheduler + concurrent access to resources
- `list.c`, `queue.c` & `tasks.c` + `croutine.c`
- `portable/GCC/ARM_CM3/port*` specific to each architecture (memory allocation and handling)

[7] R. Barry, *FreeRTOS on RISC-V*, FOSDEM 2019 à `https://fosdem.org/2019/schedule/event/riscvfreertos/`

[8] R. Barry, *Using the FreeRTOS Real Time Kernel – a Practical Guide, Cortex M3 Edition*, (2010)

Example of tasks in FreeRTOS

```
#include "stm32f10x.h"    // hardware specific
#include <stm32/gpio.h>
#include "FreeRTOS.h"     // FreeRTOS specific
#include "task.h"
#include "queue.h"
#include "croutine.h"

int main(void){
 Led_Init();
 Usart1_Init();
 xTaskCreate( vLedsFloat, (signed char*) "LedFloat",100,NULL,10,NULL );
 xTaskCreate( vLedsFlash, (signed char*) "LedFlash",100,NULL,10,NULL );
 xTaskCreate( vPrintUart, (signed char*) "Uart",    100,NULL,10,NULL );
 vTaskStartScheduler();
 while(1);                // should never be reached
}

void vLedsFloat(void* dummy)  // chaque tache est infinie
{while(1){
  GPIO_SetBits (GPIOC, GPIO_Pin_2);  vTaskDelay(120/portTICK_RATE_MS);
  GPIO_ResetBits(GPIOC, GPIO_Pin_2); vTaskDelay(120/portTICK_RATE_MS);
 }
}

void vLedsFlash(void* dummy)
{while(1){
  GPIO_SetBits (GPIOC, GPIO_Pin_1); vTaskDelay(301/portTICK_RATE_MS);
  GPIO_ResetBits(GPIOC, GPIO_Pin_1);vTaskDelay(301/portTICK_RATE_MS);
 }
}

void vPrintUart(void* dummy) // Writes each 500 ms
{portTickType last_wakeup_time;
 last_wakeup_time = xTaskGetTickCount();
 while(1){uart_puts("Hello World\r\n");
    vTaskDelayUntil(&last_wakeup_time, 500/portTICK_RATE_MS);
  }
}
```

`BaseType_t xTaskCreate(code, name, stack, parameters, priority, handler);`

Passing parameters to tasks

- A variable on the stack will be lost once the scheduler is launched ⇒ passed value is erroneous.
- **Store the passed variable in RAM** (on the heap) by prefixing its declaration with `static`

```
void func(void* p)
{ int numero= *(int*) p;
  uart_putc('a'+numero); vTaskDelay(500 / portTICK_RATE_MS);
  while (1) { vTaskDelay(100 / portTICK_RATE_MS); };
}

int main()
{
//www.freertos.org/FreeRTOS_Support_Forum_Archive/February_2007/→
      ↪freertos_Problems_with_passing_parameters_to_task_1666309.html
  static int p[5]={0,1,2,3,4};
  static char * taskNames[5] = {"P0","P1","P2","P3","P4"};

  int i;
  Led_Init();
  Usart1_Init();
  for (i=0;i<NB_PHILO;i++)
      {xTaskCreate(func, taskNames[i], STACK_BYTES(256), (void*)&p[i],1,0);}
  vTaskStartScheduler();
  while(1);
  return 0;
}
```

Type `void*`: unspecified memory area pointer, must be cast to the kind of variable being sent to the task in the called function – could be a `structure` for multiple elements.

void*

Different argument types can be provided to a function (same prototype)

```
#include <stdio.h>
void my_fonction(void* param,int type)
{long *l;
 float *f;
 if (type==1) {l=(long*)param; printf("%x ",*l);}
 if (type==2) {f=(float*)param; printf("%f",*f);}
}

int main()
{long l=0x42; float f=42.42;
 my_fonction(&l,1); my_fonction(&f,2);
}
```

```
$ gcc -Wall -o t t.c
$ ./t
42 42.419998
```

and for multiple arguments

```
#include <stdio.h>
struct my_param {long p1;long  p2;long p3;long p4;};

void my_fonction(void* param)
{struct my_param *p=(struct my_param*)param;
 printf("%lx %lx %lx %lx\n",p->p1,p->p2,p->p3,p->p4);
}

int main()
{struct my_param p={.p1=0x42,.p2=0x43,.p3=0x44,.p4=0x45};
 my_fonction(&p);
}
```

```
$ gcc -Wall -o t t.c
$ ./t
42 43 44 45
```

Memory allocation

"The FreeRTOS.org heap is the area of memory you allocated for use by the kernel. When a task is created the memory used by the task comes from the heap (the task stack and the TCB [9] structure). Likewise when you create a queue the memory used by the queue comes from the heap."

R. Barry [10]

In FreeRTOSConfig.h:

```
#define configTOTAL_HEAP_SIZE ((size_t) ( 5* 1024 ))
```

defines the total amount of memory allocated to all tasks

```
BaseType_t xTaskCreate(    TaskFunction_t pvTaskCode,
                           const char * const pcName,
                           configSTACK_DEPTH_TYPE usStackDepth,
                           void *pvParameters,
                           UBaseType_t uxPriority,
                           TaskHandle_t *pxCreatedTask
                       );
```

"If a task is created using xTaskCreate() then the required RAM is automatically allocated from the FreeRTOS heap." [11]

[9] Thread Control Block

[10] https://www.freertos.org/FreeRTOS_Support_Forum_Archive/January_2007/freertos_About_Stack_and_Heap_1642498.html

[11] https://www.freertos.org/a00125.html

Displaying the list of tasks

- $\simeq$ `ps aux` on Unix: list of tasks, used resources and status
- `vTaskList` requires an array of about 40 characters per task to be displayed
- function documentation available on the FreeRTOS web site at `https://www.freertos.org/a00021.html#vTaskList` ...
- ... where we learn that activating the function requires `configUSE_TRACE_FACILITY` and `configUSE_STATS_FORMATTING_FUNCTIONS` to be set to 1 in `FreeRTOSConfig.h`
- ... and the meaning of the status flag ("B"locked, "R"eady, "D"eleted, "S"uspended) [12]

```
Hello World
Uart            R       4       301     3
IDLE            R       0       0       4
LedFlash        B       4       242     2
LedFloat        B       4       242     1
```

Try changing the stack allocation for each task and observe the consequence

[12] FreeRTOS functions starting with `v` are procedures that do not return arguments. A function starting with `x` will return a result or handle.

Task priority

- A low priority task can be preempted by a high priority task but not the opposite.
- We have defined three tasks, two blinking LEDs and one displaying a message on the serial port. Try including an infinite loop `while(1) {};` in one of the LED blinking tasks and setting its priority high ⇒ consequence on the blinking LED ?

```
xTaskCreate( vLedsFloat, (signed char*) "LedFloat",64,NULL,1,NULL );
xTaskCreate( vLedsFlash, (signed char*) "LedFlash",64,NULL,2,NULL );
```

- switch priorities ⇒ consequence on the blinking LED ?

```
xTaskCreate( vLedsFloat, (signed char*) "LedFloat",64,NULL,2,NULL );
xTaskCreate( vLedsFlash, (signed char*) "LedFlash",64,NULL,1,NULL );
```

Stack corruption

- Classical problem with baremetal C programming: stack and heap collision, or stack corruption due to invalid pointer handling
- `#define configCHECK_FOR_STACK_OVERFLOW 2` requires the definition of the callback function `void vApplicationStackOverflowHook(TaskHandle_t xTask, signed char *pTaskName)` to notify of stack corruption
- The *idle* is allocated `configMINIMAL_STACK_SIZE` bytes on the stack.

Activating the stack corruption mechanism and setting `configMINIMAL_STACK_SIZE` to 10 will lead to stack overflow:

```
Stack: IDLE
Stack: IDLE
Stack: IDLE
```

Stack corruption

FreeRTOS can tell us how much space is left on the stack:

1. activate `uxTaskGetStackHighWaterMark(NULL);` (NULL means that a task monitors its own state) ...
2. .. and the option `#define INCLUDE_uxTaskGetStackHighWaterMark 1` in the configuration file

Message queues between tasks

Sharing messages between tasks – try adding tasks inducing more delays

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "common.h"
xQueueHandle qh = 0;

void task_tx(void* p)
{int myInt = 0;
  while(1)
    {myInt++;
     if (!xQueueSend(qh, &myInt, 100))
         uart_puts("Failed to send item to queue within 500ms");
     vTaskDelay(1000);
    }
}

void task_rx(void* p)
{char c[10];
 int myInt = 0;
 while(1)
    {if (!xQueueReceive(qh, &myInt, 1000))
       uart_puts("Failed to receive item within 1000 ms");
     else {c[0]='0'+myInt;c[1]=0;
           uart_puts("Received: ");uart_puts(c);uart_puts("\r\n");
          }
    }
}

int main()
{Led_Init();
 Usart1_Init();
 qh = xQueueCreate(1, sizeof(int));
 xTaskCreate(task_tx, (signed char*)"t1", (128), 0, 2, 0);
 xTaskCreate(task_rx, (signed char*)"t2", (128), 0, 2, 0);
 vTaskStartScheduler();
 return 0;
}
```

Practical demonstration

FreeRTOS on STM32 : `github.com/jmfriedt/tp_freertos` and lab session syllabus at `jmfriedt.free.fr/tp_freertos.pdf` [in French]

- using the cross-compilation toolchain (`arm-...-gcc`) and tools for transferring the binary file to the microcontroller (baremetal C examples)
- specificity of the STM32 architecture: high granularity of the clock configuration
- 32 bit microcontroller able to run an executive environment
- many version of this microcontroller core: the *linker* script defines what memory resources are available (`ld/` directory).
- No hardware abstraction layers: write your own low level library relying on the core manual [13] and processor datasheet [14]
- Download (git clone –recursive) FreeRTOS at `github.com/FreeRTOS/FreeRTOS-LTS`
- Tune example Makefiles to link with this FreeRTOS source tree (paths)
- runs with qemu (`https://github.com/beckus/qemu_stm32`):

Table 5.High-density STM32F103xx pin definitions (continued)

Pins						Pin name	Type(1)	I / O Level(2)	Main function(3) (after reset)	Alternate functions(4)	
LFBGA144	LFBGA100	WLCSP64	LQFP64	LQFP100	LQFP144					Default	Remap
K1	H1	-	-	20	31	V_{REF-}	S		V_{REF-}		
L1	J1	F7 (8)	-	21	32	V_{REF+}	S		V_{REF+}		
M1	K1	G8	13	22	33	V_{DDA}	S		V_{DDA}		
J2	G2	F6	14	23	34	PA0-WKUP	I/O		PA0	WKUP/USART2_CTS(9) ADC123_IN0 TIM2_CH1_ETR TIM5_CH1/TIM8_ETR	
K2	H2	E6	15	24	35	PA1	I/O		PA1	USART2_RTS(9) ADC123_IN1/ TIM5_CH2/TIM2_CH2(9)	
L2	J2	H8	16	25	36	PA2	I/O		PA2	USART2_TX(9)/TIM5_CH3 ADC123_IN2/ TIM2_CH3 (9)	
M2	K2	G7	17	26	37	PA3	I/O		PA3	USART2_RX(9)/TIM5_CH4 ADC123_IN3/TIM2_CH4(9)	
G4	E4	F5	18	27	38	V_{SS_4}	S		V_{SS_4}		
F4	F4	G6	19	28	39	V_{DD_4}	S		V_{DD_4}		

```
qemu_stm32/arm-softmmu/qemu-system-arm -M stm32-p103 -serial stdio \
    -serial stdio -serial stdio -kernel usart_cm3.bin
```

[13] `http://www.st.com/web/en/resource/technical/document/reference_manual/CD00171190.pdf`

[14] `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/CD00191185.pdf`