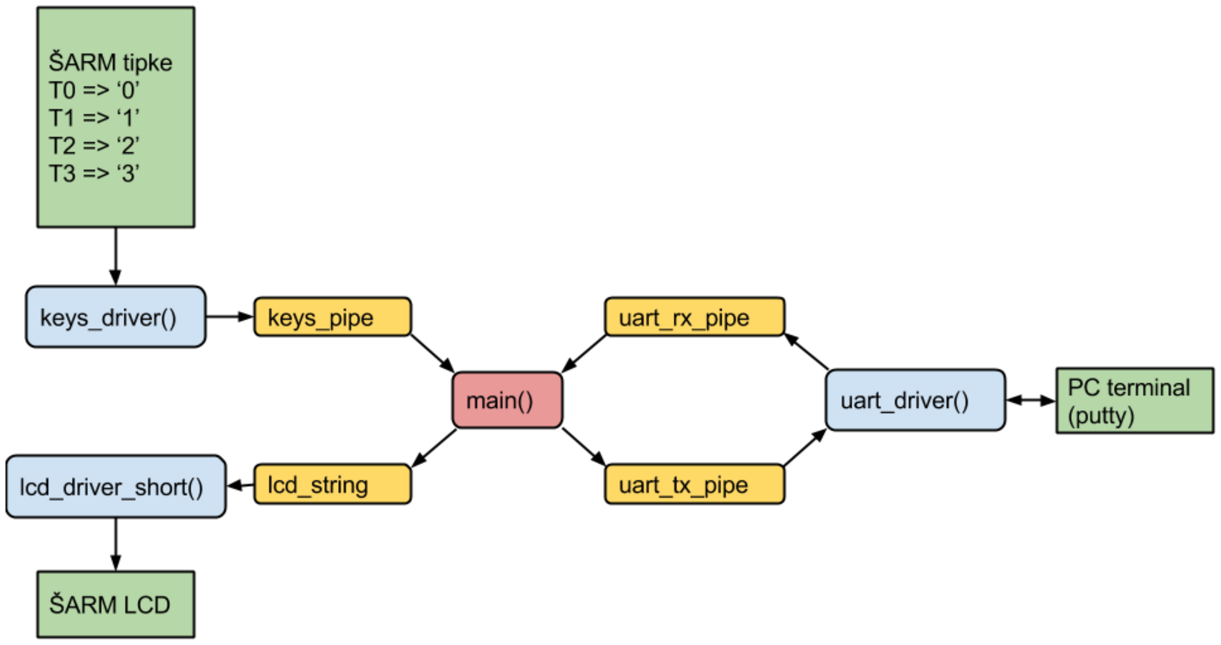
Task 7 (advanced RTOS2)

In this exercise you will learn about a slightly more advanced real-time operating system, which eliminates several limitations of the simple operating system used in previous exercises.

1. Write an UART driver, that will be able to receive and transmitt 8 bit data. The driver should store the received data into a receiver FIFO (e.g. ***uart_rx_pipe***), which is then available to the other parts of the application as a source of received data. At the same time, the driver should also check and send any data in the outgoing FIFO (e.g. ***uart_tx_pipe***), which is used by the other parts of the application to transmit data.

2. Use the upgraded RTOS2 to write a program that will display the pressed characters from PC terminal (PUTTY) on the ŠARM LCD. The pressed ŠARM buttons should also be displayed on the PC terminal (PUTTY).

The flowchart of the program is depicted in the figure below.



Using the upgraded RTOS2

The source code of the new real-time operating system resides in files with rtos2 prefix (rtos2*.c and rtos2.h). The start-up code in crt0.s file is also modified regarding the start-up code used with simple operating system.

After the start-up code we find ourselves in a function named *start_up()*, which is meant for initialisations. It is located in the startup.c file. First you have to initialise the microcontroller, raise the operating system, etc. Continue in main.c file with source code of the main program. The tasks, which will be handeled by the RTOS, can be placed in the main.c file or in a separate module (e.g. rtos_tasks.c, rtos_tasks.h). We will need the lcd driver task, the driver for the keys and the UART driver code from our previous excercises. Copy the code to this new project.

Features of the upgraded RTOS2:

- tasks are scheduled by priorities,
- time interval between two subsequent calls is defined for each task separately,
- task duration is not limited, although the tasks have to have a definite length,
- interrupts can be used normally, an exception is timer0 interrupt, which is used by the operating system, and
- dynamic task and data pipe management and dynamic memory allocation.

The operating system is managed by its functions declared in rtos2.h. Therefore, to use the functions an include directive of rtos2.h is needed.

Operating system is initialised by *rtos2init()* function. Function declaration is:

*void rtos2init(char *memory, unsigned int size, unsigned int slice);*

-*memory*	...pointer to the system memory chunk,
-*size*	...size of the system memory chunk in bytes, and
-*slice*	...timeslice in microseconds.

System memory chunk is a chunk of RAM, which is dynamically used by the operating system. Its size depends on number of created tasks, number of created pipes and pipe sizes. It must be defined as a global array.

When function *rtos2init()* ends the operating system does not run yet. It is raised by calling *enable_os()* function. Likewise the operating system can be brought to a halt by calling *disable_os()* function. When stopped, the operating system can be reinitialised.

void enable_os();
void disable_os();

During the operating system initialisation (*rtos2init()*), a configuration of timer0 interrupt is done. The system uses timer0 to measure its timeslice. For this reason function vic_init() must not be called after the *rtos2init().* In other words, all other interrupts used in the application must be configured before the *rtos2init()* call. The operating system uses timer0 configured as an IRQ with the highest priority. This means, that slot 0 must not be used at interrupt configuration by vic_init() function.

Example:

char system_memory[500];	// Global array representing system memory chunk of 500 bytes.
...	
vic_init(...);	// Interrupt configuration with exception of timer0 on IRQ slot 0.
rtos2init(system_memory, 500, 10000);	// Operating system initialisation with 10ms timeslice.
enable_os();	// Raise the operating system.
...	
disable_os();	// Bring operating system to a halt.

Data pipes (FIFO buffers)

When the operating system is initialised data pipes can be created. The tasks use data pipes for data transfer. Therefore pipes should be created before tasks. A data pipe is created by *rtos2pipe_create()* function, which returns a pointer to a pipe. Function declaration is:

*struct rtos2pipe *rtos2pipe_create(unsigned int size);*

-*size*...pipe buffer size in bytes.

Functions *rtos2pipe_write()* and *rtos2pipe_read()* are used to write data to a pipe and to read from one. Both functions return number of bytes written into a pipe and number of bytes read from a pipe respectively. Data cannot be written to a pipe when pipe's buffer is full. And it cannot be read from a pipe when the buffer is empty.

*unsigned int rtos2pipe_write(struct rtos2pipe *pipe, char *data, unsigned int size);*

-*pipe*...pointer to a pipe to write to,
-*data*...pointer to the beginning of data to be written, and
-*size* ...number of bytes in *data* to be written.

*unsigned int rtos2pipe_read(struct rtos2pipe *pipe, char *buffer, unsigned int size);*

-*pipe* ...pointer to a pipe to read from,
-*buffer*...pointer to an array, where read data will be stored, and
-*size* ...number of bytes to be read to *buffer*.

Data pipe no longer in use can be destroyed by *rtos2pipe_delete()* function.

```
void rtos2pipe_delete(struct rtos2pipe *pipe);
```

-*pipe*...pointer to a pipe to be destroyed.

Example:

```
char buf[10];
unsigned int n;
struct rtos2pipe *transfer;
  ...
transfer = rtos2pipe_create(100);        // Create a pipe with buffer of 100 bytes.
  ...
n = rtos2pipe_write(transfer, buf,       // Try to write 10 bytes to pipe transfer from array
10);                                     buf.
  ...
n = rtos2pipe_read(transfer, buf, 10); // Try to read 10 bytes from pipe transfer to array buf.
  ...
rtos2pipe_delete(transfer);              // Destroy the pipe transfer.
```

Pipes can be used in tasks and in the main program. But they cannot be used inside an interrupt request code (IRQ and FIQ). For that purpose the *rtos2pipe_...()* functions (see rtos2pipe.c file) must be modified. To enable usage of pipes inside the interrupt request code the interrupts must be temporarily disabled during critical parts of code. That can be achieved by setting (at critical part start) and withdrawing (at critical part end) I flag (to disable IRQs) and F flag (to disable FIQ), respectively.

Tasks

Task is a function, which returns nothing and receives an arbitrary pointer. Such a function is defined as an *rtos2taskptr* type. A task is created by *rtos2task_create()* function and destroyed by *rtos2task_delete()* function.

```
typedef void (* rtos2taskptr)(void *);
void rtos2task_create(rtos2taskptr function, void *arg, unsigned int priority, unsigned int
interval);
```

-*function*...task function,
-*arg* ...an arbitrary pointer to be passed to the task,
-*priority* ...task priority (lower value means higher priority), and
-*interval* ...number of timeslices in which the task is scheduled once.

The last argument *interval* defines the frequency of task calls. The task will be scheduled once per time interval *t* = *interval* * *timeslice*. Operating system timeslice length was defined at system initialisation by argument *slice* in function *rtos2init()*.

void rtos2task_delete(rtos2taskptr function);

-*function*...function of the task to be destroyed.

Each task is regularly scheduled with the frequency given at task creation. If two or more tasks are scheduled for the same timeslice then the task with the highest priority is started. If a higher priority task is scheduled during another lower priority task execution then the lower priority task is interrupted by higher priority task. The lower priority task resumes its execution when the higher priority task finishes. In other words a lower priority task waits for its turn until there are no requests for higher priority tasks. Therefore a task with low priority can miss its timeslice when its execution should start. It will be started one or a few timeslices later when higher priority tasks will be handled. But starting delay does not mean a delay at next low priority task scheduling. The delays do not sum. The frequency of task calls remains constant. Each task, regardless its priority, is scheduled once per its time interval.

Example:

```
char arg[10];
void task(void *data);
  ...
rtos2task_create(task, arg, 5, 100);   // Function task(arg) will be scheduled in every 100th timeslice.
  ...
rtos2task_delete(task);                // Destroy the task.
```

Dynamic memory allocation

A chunk of RAM can be assigned for dynamic memory allocation purposes by *rtos2mem_create()* function. Such a memory chunk is also called a memory region. It is possible to define several independent memory regions.

*void rtos2mem_create(struct rtos2mem *region, char *memory, unsigned int size);*

-*region* ...pointer to a memory region (a structure with memory region information),
-*memory*...pointer to a memory chunk (a chunk of RAM defined as global array), and
-*size* ...size of memory chunk in bytes.

Memory within a region is dynamically allocated and released by *rtos2mem_allocate()* and *rtos2mem_free()* functions. Function *rtos2mem_allocate()* returns a pointer to allocated memory. If memory allocation fails, it returns 0x00000000.

*void *rtos2mem_allocate(struct rtos2mem *region, unsigned int size);*

-*region*...pointer to memory region, where memory will be allocated, and
-*size* ...number of bytes to be allocated.

*void rtos2mem_free(struct rtos2mem *region, void *pointer);*

-*region* ...pointer to memory region, where allocated memory will be released, and

-pointer...pointer to allocated memory.

A particular memory region should be accessed by only one task (main program, IRQ or FIQ). If that is not so, the simultaneous access problem occurs. Special care must be taken to avoid memory region corruption because of simultaneous access. It must be assured that a particular memory region is accessed by only one task (main program, interrupt request code) at a time. A task (main program, interrupt request code) accessing a memory region (allocating, using or releasing a memory within the region) must not be interrupted by another task (interrupt request code) accessing the same memory region. The problem can be solved by memory locking, which means that critical parts of code must be written in a special way. On the other hand we can use *disable_os()* and *enable_os()* functions to temporarily disable the operating system during the critical parts (to prevent interruption by another task), and I and F flags to temporarily disable IRQs and FIQ (to prevent interruption by new interrupt requests).

Operating system has its own memory region defined at initialisation.

Example:

```
void *ptr;
char chunk[1000];
struct rtos2mem piece;
  ...
rtos2mem_create(&piece, chunk,       // Create memory region piece of 1000 bytes.
1000);
  ...
ptr = rtos2mem_allocate(&piece, 100); // Allocate 100 bytes in piece memory region,
                                      // ptr points to allocated memory.
  ...
rtos2mem_free(&piece, ptr);           // Release memory in piece region on which ptr
                                      points
```