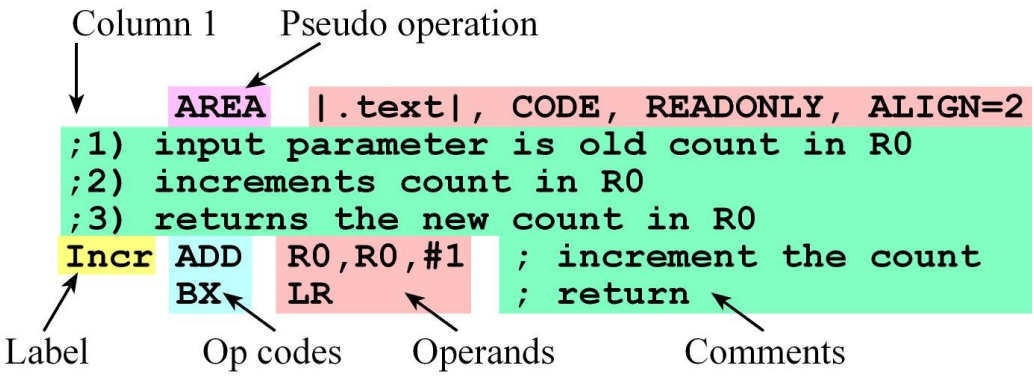
UTAustinX: UT.RTBN.12.01x Realtime Bluetooth Networks

Ассемблер ARM Cortex-M



Режимы адресации

Immediate	Data within the instruction	MOV R0,#1
Indexed	Data pointed to by register	LDR R0,[R1]
Indexed with offset	Data pointed to by register	LDR R0,[R1,#4]
PC-relative	Location is offset relative to PC	BL Incr
Register-list	List of registers	PUSH {R4,LR}

No addressing mode: Some instructions operate completely within the processor and require no memory data fetches. For example, the ADD R1,R2,R3 instruction performs R2+R3 and stores the sum into R1.

Инструкции:

LDR Rd, [Rn] ; load 32-bit memory at [Rn] to Rd
STR Rt, [Rn] ; store Rt to 32-bit memory at [Rn]
LDR Rd, [Rn, #n] ; load 32-bit memory at [Rn+n] to Rd
STR Rt, [Rn, #n] ; store Rt to 32-bit memory at [Rn+n]

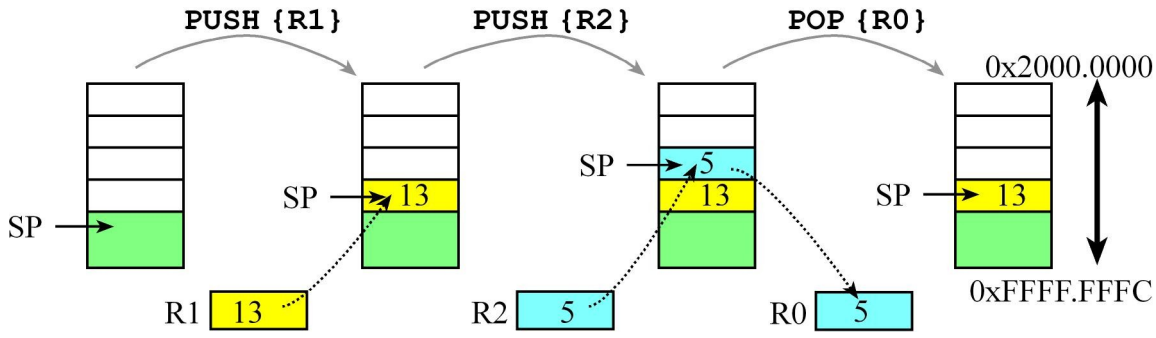
MOV Rd, Rn ;Rd = Rn
MOV Rd, #imm12 ;Rd = M
ADD Rd, Rn, Rm ;Rd = Rn + Rm
ADD Rd, Rn, #imm12 ;Rd = Rn + M
SUB Rd, Rn, Rm ;Rd = Rn - Rm
SUB Rd, Rn, #imm12 ;Rd = Rn - M
CPSID I ;disable interrupts, I=1
CPSIE I ;enable interrupts, I=0

B label ;branch to label
BX Rm ;branch indirect to location specified by Rm
BL label ;branch to subroutine at label

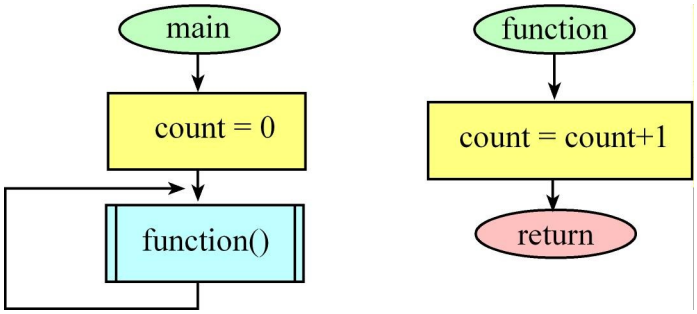
Работа со стеком:

PUSH {Rn,Rm} ; push Rn and Rm onto the stack
PUSH {Rn-Rm} ; push all registers from Rn to Rm onto the stack
POP {Rn,Rm} ; pop two 32-bit numbers off stack into Rn, Rm
POP {Rn-Rm} ; pop multiple 32-bit numbers off stack to Rn through Rm

the registers are stored in memory such that the register with the smaller number is stored at the address with a smaller value



Пример программы:



```
  AREA DATA
count SPACE 4 ; 32-bit data
  AREA |.text|,CODE,READONLY,ALIGN=2
function
  LDR R0,=count ;5
  LDR R1,[R0] ;6 value of count
  ADD R1,#1 ;7
  STR R1,[R0] ;8 store new value
  BX LR ;9
Start LDR R0,=count ;1
  MOV R1,#0 ;2
  STR R1,[R0]  ;3 store new value
loop BL function  ;4
  B loop ;10
```

```
uint32_t count

void function(void){
  count++; // 5,6,7,8
} // 9


int main(void){
  count = 0; // 1,2,3
  while(1){
    function(); // 4
  } // 10
}
```

The ARM Architecture Procedure Call Standard, **AAPCS**, part of the ARM Application Binary Interface (**ABI**):

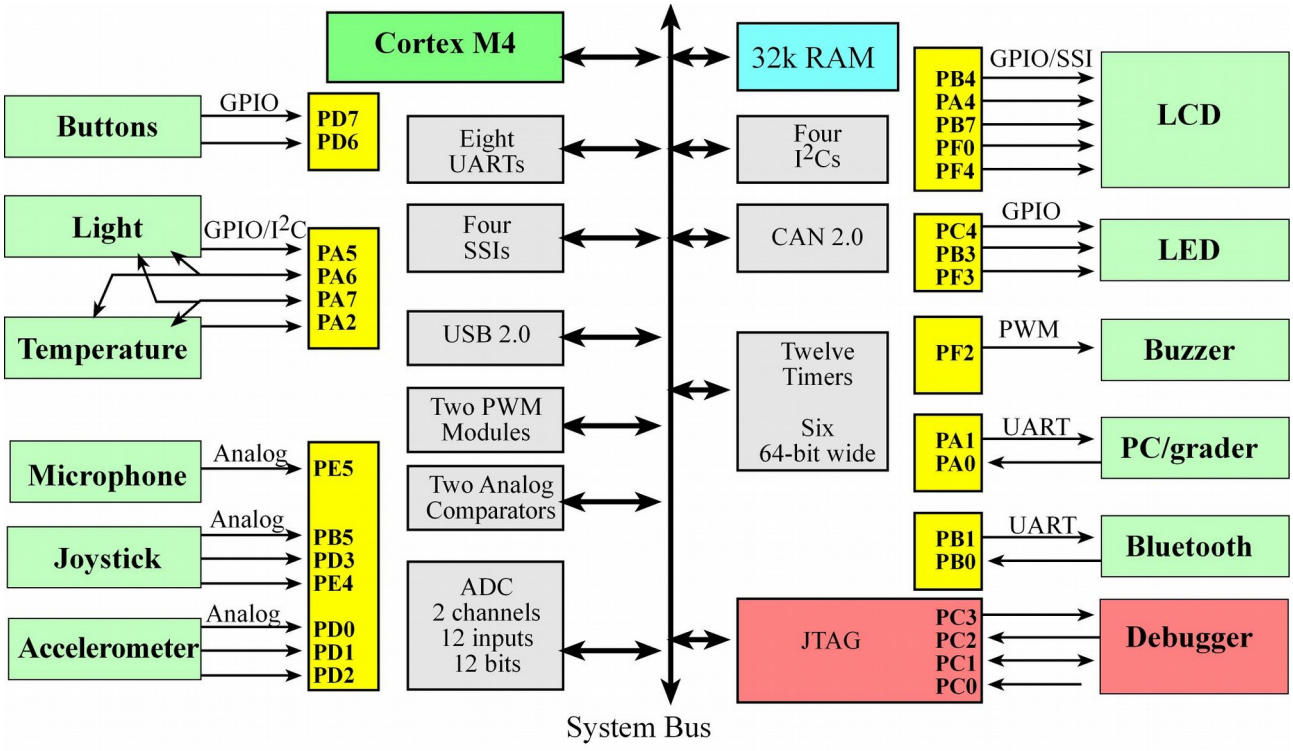
- uses registers R0, R1, R2, and R3 to pass input parameters into a C function
- functions must preserve the values of registers R4–R11
- according to AAPCS we place the return parameter in Register R0
- AAPCS requires we push and pop an even number of registers to maintain an 8-byte alignment on the stack

ARM's Cortex Microcontroller Software Interface Standard (**CMSIS**) is a standardized **hardware abstraction layer** for the Cortex-M processor series:

- **CMSIS-CORE**: API for the Cortex-M processor core and peripherals. It provides at standardized interface for Cortex-M0, Cortex-M3, Cortex-M4, SC000, and SC300. Included are also SIMD intrinsic functions for Cortex-M4 SIMD instructions.
- **CMSIS-DSP**: DSP Library Collection with over 60 Functions for various data types: fixed-point (fractional q7, q15, q31) and single precision floating-point (32-bit). The library is available for Cortex-M0, Cortex-M3, and Cortex-M4. The Cortex-M4 implementation is optimized for the SIMD instruction set.
- **CMSIS-RTOS API**: Common API for Real-Time operating systems. It provides a standardized programming interface that is portable to many RTOS and enables software templates, middleware, libraries, and other components that can work across supported RTOS systems.
- **CMSIS-SVD**: System View Description for Peripherals. Describes the peripherals of a device in an XML file and can be used to create peripheral awareness in debuggers or header files with peripheral register and interrupt definitions.

BSP

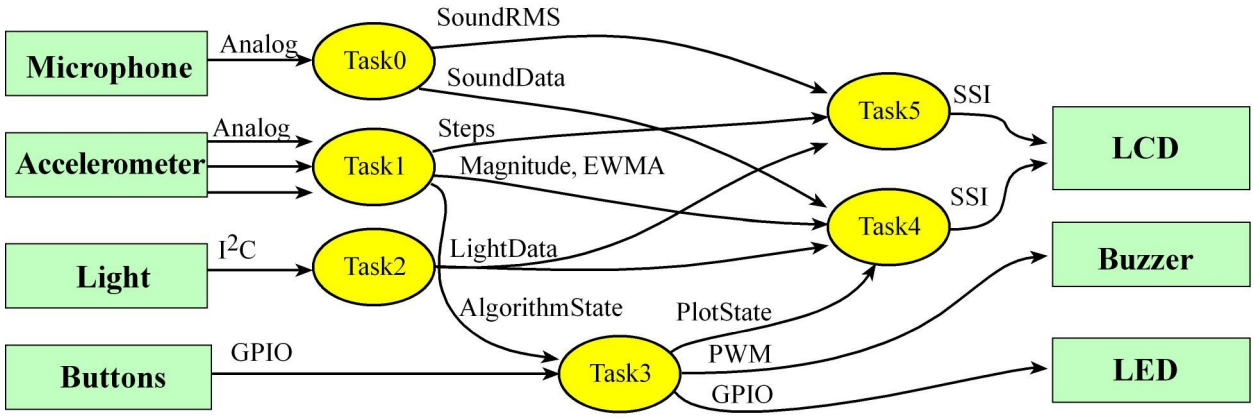
One of the important resources the OS must manage is I/O. It is good design practice to provide an abstraction for the I/O layer. Three equivalent names for this abstraction are hardware abstraction layer (**HAL**), **device driver**, and board support package (**BSP**). From an operating system perspective, the goal is the make it easier to port the system from one hardware platform to another.



Проект BoardSupportPackage

BSP.h, BSP.c

Лабораторная работа №1



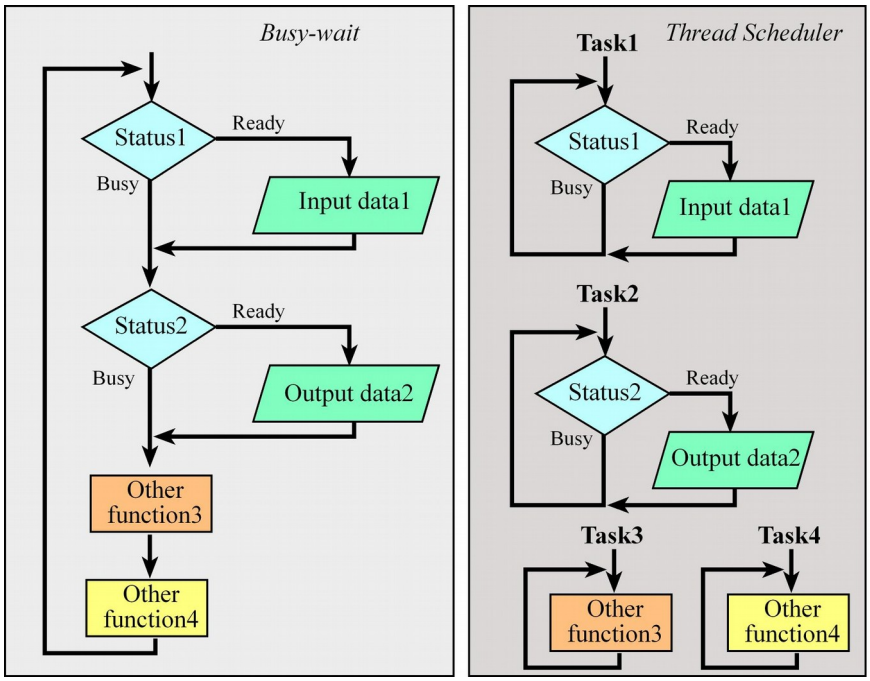
Your assignment is to increase the execution rate of Task0 from 10 to 1000 Hz, while maintaining the existing execution rates of the other five tasks. In particular, we are asking you to modify the main program, such that

•Task0 runs approximately every 1ms

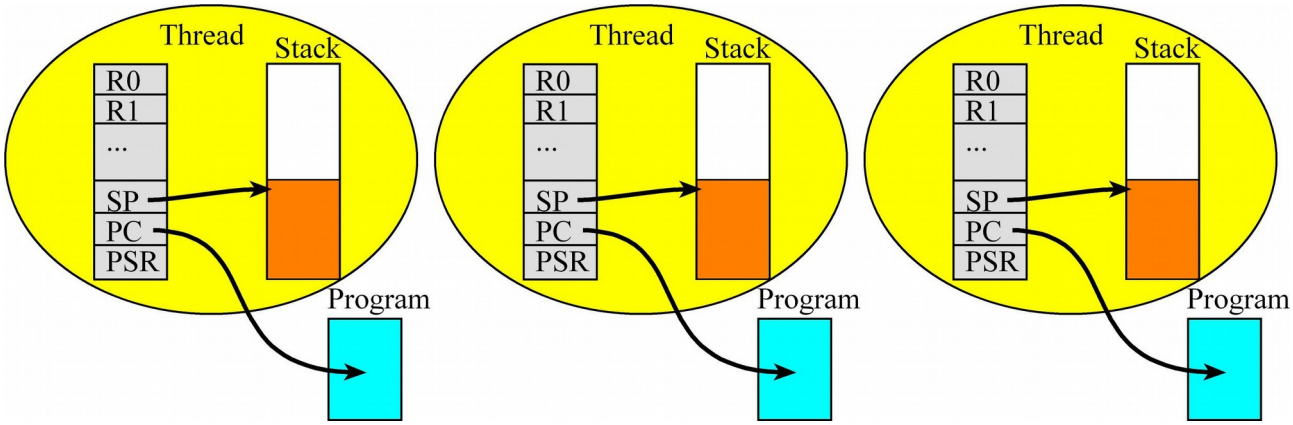
•Task1 runs approximately every 100ms
•Task2 runs approximately every 1s
•Task3 runs approximately every 100ms
•Task4 runs approximately every 100ms
•Task5 runs approximately every 1s

Управление потоками

One of the features implemented in an RTOS is a **thread scheduler**, which will run all threads in a manner that satisfies the constraints of the system.



The **thread switcher** will suspend one thread by pushing all the registers on its stack, saving the SP, changing the SP to point to the stack of the next thread to run, then pulling all the registers off the new stack.



Разделяемые потоками ресурсы:

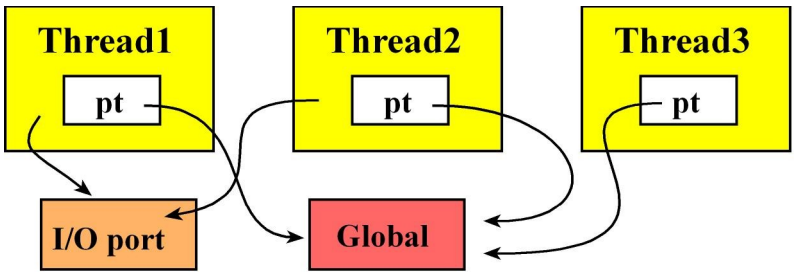
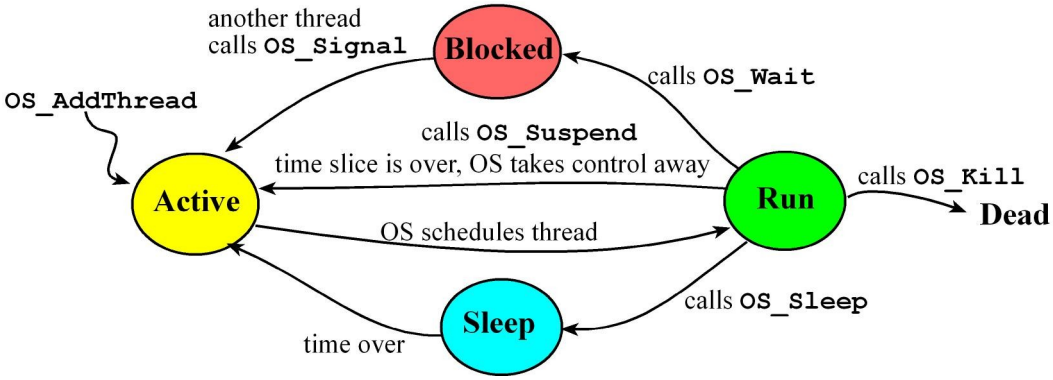


Диаграмма состояний потоков:



Timing constraints can be classified into two types. The first type is **event-response.**

Let *Ei* be the times that events occur in our system, and *Ti* be the times these events are serviced. **Latency** is defined as

$\Delta i = Ti - Ei$ for $i = 0, 1, 2, \ldots, n\text{-}1$

A second type of timing constraint occurs with **prescheduled** tasks. For example, we could schedule a task to run periodically. If we define *f*s as the desired frequency of a periodic task, then the **desired period is $\Delta t = 1/fs$**.

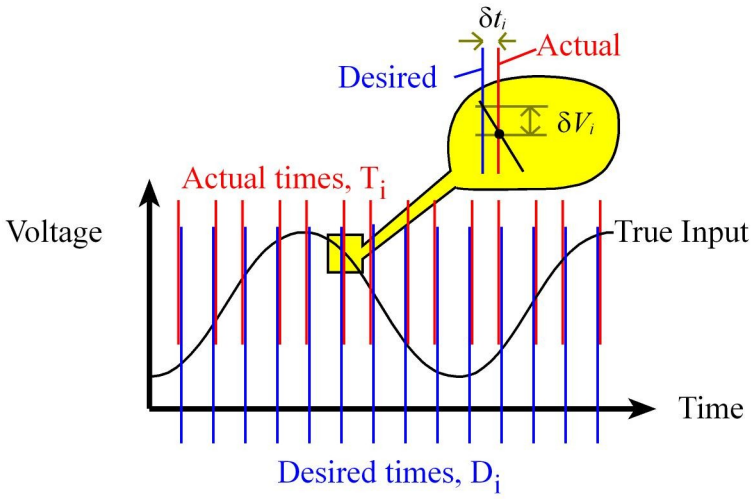
Desired time to run the *i*'th periodic instance of the task is given as

$Di = T0 + i*\Delta t$ for $i = 0, 1, 2, \ldots, n\text{-}1$

where *T0* is the starting time for the system.

Let *Ti* be the actual times the task is run, so in this case **jitter** is

$\delta ti = Ti - Di$ for $i = 0, 1, 2, \ldots, n\text{-}1$



Notice for prescheduled tasks the jitter can be positive (late) or **negative** (early).

$\delta Vi = \delta ti * dV/dt$ for $i = 0, 1, 2, \ldots, n\text{-}1$

For cases where the starting time, *T0*, does not matter, we can simplify the analysis by looking at time differences between when the task is run, $\Delta Ti = (Ti - Ti\text{-}1)$. In this case, jitter is simply

$\delta ti = \Delta Ti - \Delta t$ for $i = 0, 1, 2, \ldots, n\text{-}1$

We will classify a system with periodic tasks as real-time if the jitter is always less than a small but acceptable value. In other words the software task always meets its timing constraint. More specifically, we must be able to place an upper bound, *k*, on the time jitter.

$-k \leq \delta ti \leq +k$ for all i