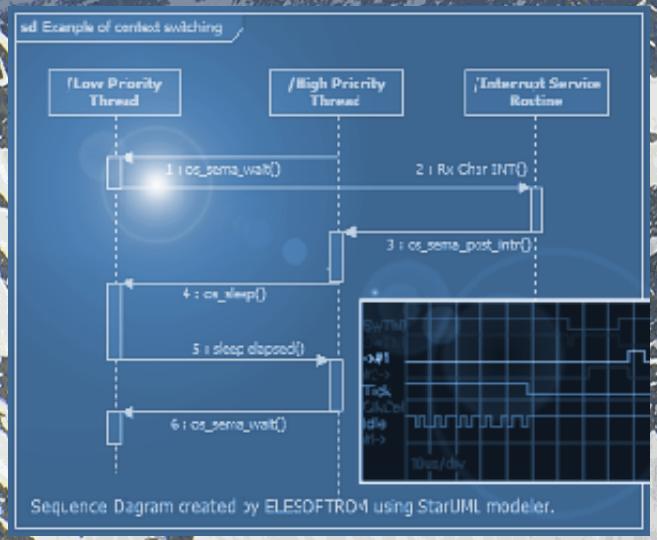
Introduction to Multithreaded Programming in Embedded Systems

Piotr Romaniuk, Ph.D.



Examples use:
- the DioneOS Real-Time Operating System,
- Code Composer Studio syntax,
- Texas Instruments msp430 microcontroller
Nevertheless, the DioneOS is available for ARM Cortex-M3 + GCC.

Contents

- Bad practices and implicated problems
- 'One-threaded' programming
- Concurrency
- Why to use multithreaded programming?
- Multithreading aspects:
  - thread and its context,
  - CPU time assignment and context switch,
  - how to control concurrency and manage shared resources,
  - periodic and deferred actions,
  - is your function reentrant?
- Testing methods

“busy-waiting” example

Naive implementation of transmission a text over UART:

```
void print_uart( char * txt)
{
  int i = 0;
  while( txt[i] )
  {
     UCA0TXBUF = txt[i];
     while( UCA0STAT & UCBUSY );
     i++;
  }
}
```

Waiting for the end of transmission of each character

- characters are written to UART buffer one after another,
- after each character, 'busy-waiting' is performed in second 'while' loop,
- until the character is transmitted processor is 'busy', execution will not return from the function, so nothing else can be done.

Problem

- print_uart() function blocks until whole text is transmitted,
- CPU time is wasted in 'busy-waiting' loop.

Solution

- avoid 'busy-waiting',
- use interrupts

Polling example

Program should do some operations and execute another function when an edge appears on selected pin on input port P1:

```
char prev = 0;
while( 1 )
{
       some_action(); //do some operations here

       char now = P1IN & TRIG_PIN;
       if( now ^ prev )
         trigger_action();
       prev = now;

       }
```

Polling the pin in each 'while' loop iteration.

Problem

- triggered action is not run immediately after the edge, but may be delayed,
- latency depends on some_action() execution time,
- some edges may be missed if intermediate processing (i.e. some_action()) is long.

Solution

- avoid polling,
- use interrupts

'One-threaded' Programming

'One-threaded' programming is very common style of programming small embedded system. This style consists in:

- splitting program jobs into 'tasks',
- critical parts triggered by interrupts are located in ISRs,
- 'tasks' are executed sequentially.

```
// Example
// Two 'tasks' are executed:
// (1) decoding commands from UART,
// (2) processing some commands:

While(1) //main loop
{
     decode_uart_cmd(); //task #1
     process_commands();//task #2
}
```

- 'Tasks' are run in 'Run-To-Completion' mode:
  - one task must wait until previous one has finished its job,
  - even if the task has nothing to do (e.g. because it waits for something), CPU time is wasted for it, at least for a call and condition check,
  - the task should return CPU control as soon as possible, otherwise it delays others.

ISR - Interrupt Service Routine

Signalling from ISR

Example: hardware timer triggers interrupt that cause ISR execution which signals this event to main loop:

```
volatile char signalling_flag = 0;
//'volatile' is required, otherwise compiler can optimize access to this variable.
 // Testing it in main loop may refer to once loaded value into register instead of
 // variable in memory. This behaviour may be affected by compiler options.

#pragma vector=TIMER0_A0_VECTOR
__interrupt void timer0_isr(void)
{
    signalling_flag = 1;
}


int main(){
...
while(1)  //main loop
{
  if( signalling_flag )   // *
  {
     signalling_flag = 0;
     execute_action(); //action executed when timer expired
  }
  ...
}
```

**) - concurrency issue is intentionally neglected here. It may happen that timer interrupt appear between positive check (by 'if') and following clear of the signalling_flag. How to prevent such problem is described in further sections about concurrency.*

ISR nesting

- check if ISR nesting is used:
  - by default, in msp430 nesting is not used, interrupts are disabled at ISR call;
  - ARM Cortex-M3 uses more complex interrupts system that involve preempting groups and subpriorities. By default, nesting is available and enabled.
- work without nesting is more popular, ISR coding is easier,
- if you use nesting set-up priorities properly,
- if interrupt services are not nested it assures that code in ISR is never interrupted,
- but it means that all pending interrupts are delayed until the exit from running ISR,

Guidelines

- keep ISR execution time short,
- do not wait in ISR,
- never block in ISR,
- in ISR, execute only time critical jobs, that require immediate response,
- defer other jobs to main loop (signal and execute it there).

Concurrency between execution in main loop and in ISR

Example: program that counts iterations in main loop between two execution of ISR.

```
volatile long counter = 0;

//in ISR:
counter=0;

//in main loop:
while(1)
{
    some_function();
    counter++;
}
```

- this is unsafe, because access to common variable (i.e. counter) is not atomic.
- one operation in C is translated into a pair of CPU instructions:

```
volatile long counter = 0; //four bytes in memory

//in ISR:
counter=0;            CLR.W   &counter
                      CLR.W   &counter+2

//in main loop:
counter++;      INC.W   &counter   <- increments the least significant word of counter,
                ADC.W   &counter+2 <- if Carry appeared, add it to the highest sign. word
```

If no extra measures are applied, interrupt can appear between these two assembler instructions in main loop.

Concurrency in 'One-threaded' Model, cont.

```
//in main loop                                              counter              Carry
                                                            0x0000 FFFF          0
counter++;

      INC.W    &counter
                                                            0x0000 0000          1
                              ---> INTERRUPT
                                   status register containing Carry flag is saved
                                  //ISR
                                  counter=0;
                                        CLR.W    &counter
                                                            0x0000 0000
                                        CLR.W    &counter+2
                                                            0x0000 0000
                              <--- CPU status register is restored on ISR exit
                                                                                 1

      ADC.W    &counter+2
                                                            0x0001 0000
```

- value of the counter is incorrect after incrementation, it should be **0x0000 0000**, or **0x0000 0001** depending on the interrupt location but is **0x0001 0000**,
- modification done in ISR is lost - it looks like last one has not happened,
- counting will be continued until next execution of ISR (next interrupt),
- this next result of counting will be incorrect, because of above issue,
- such issue will be very rare,
- it will be difficult to find it by debugging, because it is hard to repeat it,
- nevertheless it may ruin your firmware, so it must not be neglected!

Concurrency in 'One-threaded' Model, cont.

Make common modifications atomic:

```
volatile long counter = 0;

//in ISR:
counter=0;

//in main loop:
while(1)
{
     some_function();
     __disable_interrupts();
     counter++;
     __enable_interrupts(); //it is assumed here that interrupts are
                            //enabled during regular work

}
```

- interrupts are disabled for a period when common variable is modified,
- do not disable interrupts for long time, otherwise it affects interrupts latency (pending ISR would be delayed by this period)
- in above example it is assumed that interrupts are enabled during regular work; if you need more general solution, then store the interrupt state before you have disabled it and restore at the end (refer to next slide).

Concurrency in 'One-threaded' Model, cont.

If you have interrupts disabled sometimes (but ISR never exits with disabled interrupts*) you may use following guarding template:

```
volatile long counter = 0;

//in ISR:
counter=0;

//in main loop:
while(1)
{
    some_function();

      unsigned short SR_saved = __get_SR_register(); //save interrupts state
    __disable_interrupts();

    counter++;

    if( SR_saved & GIE )//restore interrupts state
         __enable_interrupts();
```

**) the condition is required because the sequence:*

```
    unsigned short SR_saved = __get_SR_register(); //save interrupts state
    __disable_interrupts();
```

is not atomic.
If the condition [*] is not true (that is very rare) other more sophisticated protection is needed.

Concurrency in 'One-threaded' Model, cont.

Other operations can be unsafe, like:

- test-modify sequence

```
if( common_variable == SOME_VALUE )
{
        common_variable += OFFSET_VALUE;
        //unexpected value may be in common_variable, one can assume that
        //  it should contain SOME_VALUE+OFFSET_VALUE
        …
}

//issue appears if the sequence is interrupted by ISR, where common_variable is modified
```

- access to information spread in a few variables (a set of data, structure fields):

```
unsigned char last_item = 0;
unsigned char items[10];

//adding an item:
items[ last_item++ ] = SOME_VALUE;
```

```
MOV.B   &last_item, R15
MOV.B   #SOME_VALUE, &items(R15)
INC.B   &last_item
```

- value of last_item is updated in last CPU instruction, if ISR would try to add an item, it will overwrite the one that is being added.
  - structure contents may be inconsistent until all modifications are finished,
  - General note: access to shared resources must be managed.

Why Multithreaded Programming?

Multithreaded programming has following advantages:

- separation of tasks, each task has its thread,
- separated parts of code for each tasks help to it keep more clear,
- easier, more flexible programming, because of threads separation,
- semi-parallel execution (step toward parallel programming*),
- loosing time dependencies unavoidable in 'one-threaded' model,
- avoiding 'busy-waiting' (effective CPU time usage),
- waiting on system objects, task is woken up when it is needed.

Still must be taken into consideration:

- concurrency issues, not only between thread and ISR but also between threads (threads are switched from one to another),
- one thread can be preempted by the operating system in any location, even if you do not call any system function (true for preemptive OS).

New requirements

- extra RAM memory needed for each thread context (own stack),
- CPU time spent on thread switch or system clock tick (check each OS for detailed data in this characteristic).

**) as long as CPU has one core it is only semi-parallel, only one program is executed in particular time.*

How to Start Multithreaded Programming?

- use operating system for your MCU,
- if you take care about your system real-time properties use Real-Time OS,
- if you work on ARM Cortex-M3 (e.g. STM32L162) or msp430 you can use the DioneOS

http://www.elesoftrom.com.pl/en/os/

The operating system provides:

- threads management including CPU time scheduling,
- necessary synchronisation objects (also useful for guarding access to shared resources),
- time dependencies support (i.e. timers, sleeping),
- communication objects,
- memory management,
- etc.

When you develop your firmware on the basis of Operating System:

- large part of the code is provided by OS vendor (standard objects, e.g. ringbuffer), you do not need to spend a time for their implementation,
- code of the OS is tested and ready to use,
- you follow well known concurrency formalism (e.g. threads, semaphores, mutexes, etc.)
- you can model your firmware and expect designed behaviour,
- if the OS is commercial, you have support from its vendor.

Example of Minimal Program in the DioneOS

Example of program in multithreaded real-time operating system (the DioneOS).
Each thread code is separated and runs in endless loop.

```
 /* ================== IDLE THREAD ============================ */
os_result_t thr_idle(void * args)
{
  while(1) //infinite loop,
  {
      debug_pin_inv( D_IDLE );//invert D_IDLE pin - only for signalling
      __delay_cycles(40);
  }
}

/* ==================== MAIN function ======================== */
int main(void)
{
  ...
  // initialization parts of the system:
  debug_pin_init(); // 1. DEBUG_PORT pins
  os_timers_init(); // 2. timers
  os_thread_init(); // 3. internal threads data
  os_thread_create( 15, thr_idle, NULL, 256); // thread creation
  os_timers_start();// start timers support, inc. system tick interrupt

  os_scheduler_run(); //run scheduler and switch the context to the thread
  //it should not return here, otherwise means that it failed to run scheduler
  while(1);
}
```

Thread code that runs in infinite loop.

System initialization and the thread creation.

Above code will initialize the system (main function) and further by calling **os_scheduler_run()** multithreaded environment is built. Threads are started, so the thread function **thr_idle()** is entered. The function should work in infinite loop and perform its jobs inside. The system will manage CPU time and schedules it for threads.
The example will generate square signal on D_IDLE pin (external output).

What is a Thread in the DioneOS ?


```
/* ================== IDLE THREAD ============================ */
os_result_t thr_idle(void * args)
{
  while(1) //infinite loop,
  {
      debug_pin_inv( D_IDLE );//invert D_IDLE pin - only for signalling
      __delay_cycles(40);
  }
}

/* ================ MAIN function ============================ */
int main(void)
{
  ...
  os_thread_create( 15, thr_idle, NULL, 256); // thread creation
  ...
}
```

Each thread has its context, i.e.:
- its own stack,
- local contents of the CPU registers.

Thread is characterized by:
- stack size,
- thread function,
- thread priority.

Note that the same code may be executed in the context of different threads.
The context determines what thread is executed.

Thread States

- In the DioneOS system the thread can be in one of following states:
  - Running – thread is in 'running' state when the CPU executes the code in its context. Because there is only one core only one thread can be in running state.
  - Ready – thread is in 'ready' state when it is ready to run, but other thread with higher priority is still running.
  - Waiting – thread is in 'waiting' state if it waits on some waiting object (e.g. semaphore, mutex or is sleeping).
- Initially thread is in 'ready' state after creation.
- After os_scheduler_run() system starts the thread with the highest priority, setting this thread to 'running' state.
- Thread state is managed by the operating system.

Scheduling the CPU Time

- The inherent part of the operating system is a scheduler.
  It is responsible for allocation the CPU time to the threads.
- The scheduler determines when to switch execution to another thread
  and to what thread it should be switched.
- The scheduler is driven by system events, like timer expiration, a release of waiting objects, etc.
- The scheduler implements a scheduling rule, that define how to order threads execution (there is a few scheduling rules, e.g. Round-Robin).
- The DioneOS uses maximum priority scheduling:
  - each thread has assigned priority number,
  - ready threads (including running one) are ordered by their priorities,
  - thread with maximum priority in set to running state.
- If the system is preemptive, like the DioneOS, it can break execution of the current thread immediately when new candidate for running state change its state to ready.
- Switching the execution from one thread to another requires a context switch.

Context Switching

- Switching from one thread to another
- Changing the context means:
  - switching to another stack,
  - switching the contents of the CPU registers
- Context switching is done in following sequence:
  - store execution point
  - store CPU flags
  - disable interrupts
  - store other CPU registers on stack
  - store SP in private thread structure
  - switch to new thread stack (load its SP)
  - restore CPU registers, including flags and interrupts state
  - jump to last executed instruction in this thread
- Context switch is atomic operation and is considered as very critical, hence it is done with disabled interrupts to provide unique access to system descriptors
- Time of context switch is a critical parameter of the RTOS, because it is an overhead of all multithreaded system; the DioneOS has been optimized for short switching time, including fast signalling from ISR by a semaphore.

The DioneOS switches in: **8 - 9.3us*** for ARM Cortex-M3 and **10-12us*** for msp430

**)- the value depends on switching conditions, for detail please visit:*

http://www.elesoftrom.com.pl/en/os/performance.php

When the Thread Can Lose the CPU?

- a consequence of context switch is losing CPU - the CPU is switched to another thread
- it may happen because of two types of events:
  i. action in running thread - calling a system function,
  ii. concurrent system event that is independent on running thread and has its origin somewhere else.

- in case [i] the switch is due the thread call:
  - the system function that blocks, e.g. os_sema_wait(),
  - the function that releases higher priority thread that has been waiting (os_sema_post())

- in case [ii] the switch is initiated by:
  - a change of higher priority thread to 'ready' state, because one of conditions:
    - releasing a waiting object on which it has been stopped (e.g. semaphore); it may be initiated by ISR (os_sema_post_intr()) or callback on timer expiration,
    - end of waiting on the object due to the timeout (os_sema_wait_timeouted())
    - elapsed sleeping time (in os_sleep())
- in case [ii] the running thread loses CPU in preemption,
- preemption can be controlled by running thread (os_preempt_disable/enable()). (it is independent on interrupt control, so when preemption is disabled ISRs can be still called)

How to Control Concurrency?

ELESOFTROM EMBEDDED SYSTEMS

Method	Can be inter-rupted by ISR?	Does it affect interrupts latency?	Can be preempted or swithed to another thread?	Function		
				Concurrency control	Threads synchronization	Access control to shared resources
Disable* interrupts	No	Yes	No	x		x [0]
Disable** preemption	Yes	No	No	x		x [1]
Semaphore^	Yes	No	Yes		x	x
Mutex^^	Yes	No	Yes			x

[0]) – use if you share resource between ISR and threads, disable only for short period
[1]) – use if you do not access resource from ISR
*) - interrupts in ISR are already disabled, nothing can preempt nor interrupt ISR
**) - used in ISR only if you want to control behaviour of preemption outside of ISR
^) - do not use functions that would block in ISR
^^) - do not use mutex in ISR, if it were locked it would block in ISR – it is general constrain

Disabling Interrupts for Critical Section


```
unsigned short flags; //local variable for storage the state of interrupts
OS_DINT_SAVE ( flags ); //saves the state and disables interrupts

 // critical section

OS_INT_RESTORE( flags ); //restore saved state of interrupts
```

- use if you need to protect the section of code from being interrupted and preempted,
- you can control in this way an access to shared resource that is also accessed from ISR,
- do not spend a lot of time in that critical section, because it disables interrupt –
  - if any interrupt happens it will be delayed, until the end of the section,
- you can nest such sections, it will handle interrupts state properly, but you need to use separated 'flags' variable for each pair of guards,
- you don't need to analyse if interrupts were enabled or disabled at the entry of this protected section. It is assured that inside interrupts are disabled and after that the state is restored,
- you must not use blocking functions inside this section.

Disabling Preemption for Critical Section


```
unsigned char preempt_state; //local variable for preemption state
preempt_state = os_preempt_disable(); //reads state of preemption and disables it
 // critical section

if( preempt_state )
  os_preempt_enable(); //restore saved state of preemption
```

- use if you need to protect the section of code from being preempted by another thread,
- NOTE that interrupts can still happen inside the section,
- in above way, you can control an access to shared resource that is accessed from threads but not from ISR,
- it does not affect interrupt latency because interrupts remain unchanged, it only controls preemption,
- use preemtion control if you need to make a few things but one of them would make preemption (e.g. you need to release two semaphores at once).
- you must not use blocking functions inside this section.

Threads Synchronization by the Semaphore


```
os_sema_t sem1;  //semaphore structure
//initialisation
os_sema_init( & sem1, 0 );  //initialised in locked state, initial count = 0

//thread #1, trigger            // thread #2, waiter
os_sema_post( & sem1 );           os_sema_wait( & sem1 );
//triggering from ISR
os_sema_post_intr( & sem1 );
```

- use if you need to trigger the waiting thread from another one or from ISR,
- context switch to the waiter happens when it will be the highest priority from all ready threads (this in an implication of scheduler in the DioneOS system),
- example: the waiter is a commands processor that are taken from the queue.

```
os_ring_buffer_ev rb; //queue implemented as ring buffer - global

os_event_t ev;  //event to be sent
ev.signal = SOME_CMD; //example of command

//thread #1, command request        // thread #2, command processor
os_ringbuf_add_ev( &rb, ev );
os_sema_post( & sem1 );             while(1)
                                    {
                                        os_sema_wait( & sem1 );//'non-busy' waiting
                                        os_event_t ev_rec;//received event
                                        os_ringbuf_get_ev( &rb, &ev_rec );
                                          //process event ev_rec, e.g.
                                        if( ev_rec.signal == SOME_CMD )
                                              //make some action
                                    }
```

You avoid 'busy-waiting' because you use waiting on the semaphore.

Access Control to Shared Resource by the Mutex


```
 os_mutex_t mtx1;  //mutex structure

 //initialization
 os_mutex_init( & mtx );  //initialized in unlocked state

void fun_access_to_resource( void )//access resource only through this function
{
 os_mutex_get( & mtx );  //lock and own mutex
 // exclusive access to shared resource - no one else can access it
 os_mutex_release( & mtx ); //releasing mutex
}
```

- mutex provides exclusive access control,
- when mutex is locked it is owned, only the owner can release it (trial of release by non-owner is considered as error, if you need to release from another thread use semaphore),
- mutex get/release functions can be nested, the system counts number of corresponding functions and release the mutex on the last matching release,
- mutex controlled section does not affect interrupts,
- do not use mutex in ISR,
- os_mutex_get() and ~release() may cause preemption,
- only one thread can access the resource.

Holding the Thread Execution

- the DioneOS system has internal clock interrupt,
- its frequency and hardware timer are selected and configured by developer,
- its period is a unit of time specified in function calls,
- if you need to hold thread execution for specific time, use os_sleep() function
- system will do other things in meantime – your waiting will not be 'busy-waiting'

```
os_sleep( 10 ); //example of waiting 10 units of internal system clock
```

- waiting on semaphore with timeout:

```
os_result_t retc;
retc = os_sema_wait_timeouted( & sem1, 10 ); //example of waiting 10 units of
                                             //internal system clock

if( OS_TIMEOUT == retc ) //waiting finished due to timeout elapsing
   ...
if( OS_STATUS_OK == retc ) //waiting broken because of semaphore release
   ...
```

- Note that the function can return with OS_TIMEOUT status code and the semaphore will be released after that. It means that semaphore has not been released within the period specified in function call.

Periodic or Deferred Action by the Timer

- if you need periodic or deferred action use timer (it can be created as repetitive or 'one-shot' correspondingly)

```
os_timer_t tm1;//timer structure

os_result_t timer_callback_fun( void * args )//callback function
{
//do an action on timer expired
return OS_STATUS_OK;
}

...
os_timer_create( &tm1, OS_TIMER_REPETITIVE, 100, timer_callback_fun, NULL );
                        //create and arm timer
...
os_timer_delete( &tm1 );//disarm and delete timer
...
```

- when timer expires attached callback function is called,
- you can pass arguments to the callback during timer creation,
- callback is run by system clock ISR, use ISR rules inside the callback,
- do not stay in callback too long, everything is blocked!
- trigger and event or semaphore and process further in a thread.

Is Your Function Reentrant?

- in multithreaded program you need to consider reentrant property of your functions.
- function is reentrant if it can be called concurrently from multiple threads and it will not generate any issues.
- check the functions that are called from different threads and/or ISR,
- in practice, it means that it has no common (global) data required for its execution,

Example of non-reentrant function

```
char rtext[100];


char * create_msg( char * prefix_msg, int x )
{
 sprintf( rtext, "%s %d\n", prefix_msg, x );
 return rtext;
}
//only one thread can execute the function
//rtext is common for all calls.
```

Example of reentrant function

```
char* create_msg( char* res, char* prefix_msg, int x )
{
  sprintf( res, "%s %d\n", prefix_msg, x);
  return res;
}
//the buffer for result is provided by a caller
//so it is independent for each thread
//No common items are in this function.
```

- function can also be not reentrant if it uses some global resources (e.g. serial port),
- if the function requires more data, form it in structure that serve as its execution context, Provide at least such separated context for each call from different thread,
- if your function is not reentrant add proper guards (e.g. lock mutex at the entry and release it on exit).

Testing Methods

- Traditional debugger is not very useful in real-time environment. The system is dependent on external events and is required to create immediate responses. In a case of debugger, interactive mode is the most common testing method. It needs to froze the system and provide manual control over program execution for the developer by means of breakpoints, step-by-step execution and variable watching.
- In testing process of the real-time system more valuable are techniques that do not interfere the system time scale.
- They can provide observation of the system behaviour in form of trace that can be analysed later:
  - Writing to log – a textual log is created during program execution. In selected locations of the code extra instructions are inserted. They print logging messages that describe its location (line number, function, module) and interested contents in that place. Sometimes the messages has also marked by timestamps.
  - Real-time observation by logic analyser – presence of selected events in the system are shown on microcontroller pins and captured by logic analyser. Recorded digital signals represent exact time sequence of the events that happened in the system.
- The information can be also provided in aggregated form. In is any kind of statistical data like counters of events, average, min, max value, etc.
- When you test your code take into consideration multithreaded aspects: the same code can be concurrently called from different threads; multiple entries may happen.
- Do not forget about concurrency in debugging parts that you have created.

Testing Methods: Writing to Log

- logging messages can be generated in the system and sent by serial port to PC, where they are recorded in a file,
- universal textual form of the log – easy to browse, filter, search
- number of logging messages must be balanced, each generated message is an extra load to the system,
- generate only the logging messages that you need,
- logging in many locations and with many details can slow down the system and change its behaviour. This can be so significant to system internal dependencies, that an analysed bug will disappear (it is called 'Heisen-bug' from Heisenberg uncertainty analogy: when you start debugging, you observation tool influence the system, so you observe something else that original one).
- use short messages to save bandwidth of logging channel, the serial port speed and messages creation are 'bottle-necks',
- provide conditional compilation that controls debug messages, you will be able to disable it globally by one #define. Even function call, if executed in many places, may be significant load for the system.

```
main:new connection
srv_handle_ev: [0] state=NOTCON (1)
srv_handle_ev: [0] conn.192.168.0.2
main: [0]: state=WAITREQ (2)
main: [1]: state=NOTCON (1)
main: waiting for data/connection
main: [0] srv=0: data 4
srv_handle_ev: [0] state=WAITREQ (2)
srv_nandle_ev: [0]->state=RECREQ (3)
srv_received: 12 bytes
dbg_request: msg. received
00 00 13 44 00 00 23 00 00 12 99 89
dbg_request: msg_len = 12
dbg_request: msg_type = 1
srv_proces_req: [0]->state=WAITREQ (2)
main: [0]: state=WAITREQ (2)
main: [1]: state=NOTCON (1)
main: waiting for data/connection
cmd_proc_stop: msg. Received 103
cmd_proc_send_rsp: send resp. T<P.1>
send_msg_xdataL sent by srv=0
srv_send: [0]<< state=WAITREQ (2)
exec_stop: sending STOP
...
```

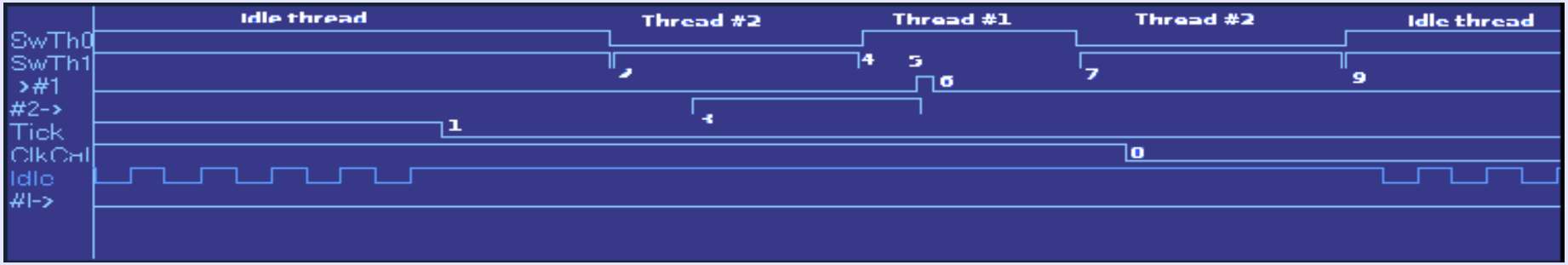
Example of log file.

- if it is possible, provide a timestamp for each message,
- assure that messages are written to the log according to they order in time,
- create logging message levels (error, warning, info, debug) and allow for conditional filtering.

Testing Methods: RT Recording by Logic Analyser

ELESOFTROM EMBEDDED SYSTEMS

- the signalling on microcontroller pins has very small impact on system behaviour,
- it can be recorder together with signals from hardware (possible latency testing, missing events, etc.) - coherent software-hardware testing
- use the logic analyser capturing method, when you need to understand exact time sequence of the events,
- use the analyser that is able to capture long period (long recording buffer),
- adjust sampling period to minimum gap width between events,
- available signalling options:
  - if you need a few events signal each one on separated line,
  - if you need more – code them as bits, capture and decode after that (e.g. 4 lines allow to signal 16 types of events)
  - if you need to signal small numbers – use set of lines (e.g. 2 bits for thread id in following figure)
  - if you need to signal larger numbers, use prefix code and partial transmission in digits.
  - use edge for signalling fast events



Example of signals captured by logic analyser: [SwTh1,SwTh0] – thread id – allows observation context switching; Tick – edge of signal represents system clock interrupt

Summary

- Develop your firmware as multithreaded. You will have more clear and well structured code. Coding will be more flexible and your tasks will be more independent.
- Use ready Real-Time Operating System. You will save a time and work, the system provides tested and ready-to-use objects designed to develop multithreaded firmware.
- Design your firmware using well known formal concepts (i.e. threads, semaphores, mutexes, etc.)
- Create a model of threads dependencies and their inter-synchronisation.
- Manage common resources that are shared between threads.
- Remember about preemption, the CPU can switch to another thread in any location.
- Consider reentrant property of common functions (avoid global variables).
- Insert in your code debugging parts, conditionally compiled.
- Use real-time methods adapted for testing this type of behaviour.
- Do not assume that something that is very rare will not happen. Ignoring such an issue would ruin your efforts toward reliable software development.