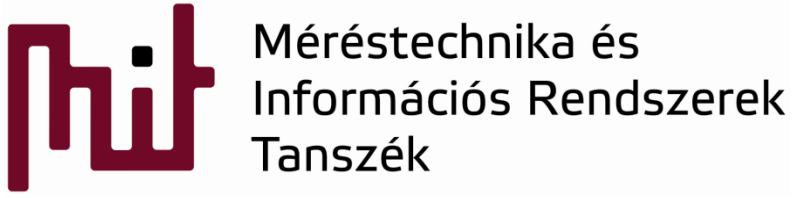
Embedded and Ambient Systems 2021.10.12.

Debugging and testing of embedded systems



Debugging

- Perfect code (especially for the first attempt) does not exist or at least is very rare
  - Some kind of testing and/or debugging are parts of the development process
- Debugging is necessary not only in the final phase of development but also important during the development process
  - The code is advised to be partitioned into subtasks and the individual tasks have to be tested separetely
  - One should not proceed with the development until all tasks are tested carefully and work as expected (if not, the problem will appear later and can cause a lot of trouble and extra work)
  - Code parts used for testing is advised to be left in the code (commented if not necessary) for later debugging

Specialties of debugging in embedded systems

- Development environment and the processor that runs the code are separated:
  - In case of PC-based debugging PC runs the debug environment and the code as well, therefore PC has direct access to the processor and the memory
  - In case of embedded systems, especially for simple processors or simple development environment the relationship of the PC and embedded system is limited to download the program
- Not only the processor but also the embedded environment has strong influence on the operation:
  - In case of a car braking system the reaction depends on the state of the brake pedal or whether the wheels are blocked or not
  - For testing a systems that processes an analogue signal (e.g. sound) the analogue signal has to be generated. Furthermore a sound cannot be generated sample per sample so it cannot be stepped
  - Consequence: not only the SW-based but also the external circumstances has to be established
    - Physical implementation is not needed all the time but the external (environmental) behavior has to be at least simulated
- Relationship between external events and bugs
- Generation of excitation signals used for examination of system behavior

Specialties of debugging in embedded systems

- Possible phases of debugging:
  - Implementation of an algorithm or operational mechanism on PC
    - E.g.: Simulink in Matlab or C code run on PC (emulation of functions corresponding to a HW handling)
  - Implementation of a function on the embedded system, simulation of the physical environment
    - In most cases some kind of prototyping fast development board is used, e.g. Giant Gecko
    - Real input and output signals are generated and measured, respectively, to test a system since some circumstances would be difficult to be established in real life
  - Testing the embedded system in real environment
    - This is the most difficult task since the real environment is not easy to be available (e.g. a plane that is falling unintentionally)

Specialties of debugging in embedded systems

- During debugging the system is “disturbed”: timings are changed
- Resources limit the debugging possibilities
  - E.g. printing variables are resource- and time-consuming

In-circuit debugging

- In-circuit emulation / debugging:
  - Debugging is possible in the final circuit environment, without removing the processor
  - Nowadays it is not so special but some decades ago it was difficult
  - A dedicated interface and internal complimentary circuitry is needed to get access to peripherals / registers / memory
  - Important parts of the system:
    - Debug interface that provides access to the processor
    - Debug circuitry to handle the debug interface
      - On some development boards it is readily available, but in a few cases it has to be bought as a separate circuit
    - PC-based development environment to handle the debugger, its functions and services

Categories of debug tools

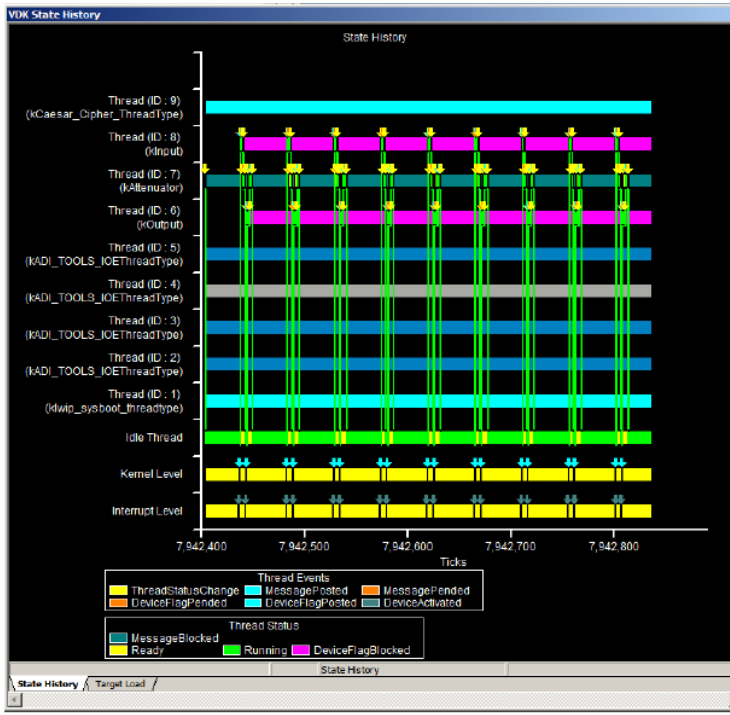
- Debugger:
  - Search for static errors
  - Running of program must be stopped to examine the processor status
  - Typical examinations:
    - Value of variables
    - Value of registers
    - Status of program counter (which part of the program is executed)
  - Tools:
    - Breakpoint (stops the program execution at a certain line)
    - Stepped execution
      - Difficulty: in case of code optimization applied, execution of a command line not necessarily follow the C-language code sequence of the originally written code

Categories of debug tools

- Tracer
  - Offers the examination of the program execution as a function of time
    - Execution of program, series of instructions
    - Following the value of a certain variable continuously
    - Saving register content
  - Data logging can be a part of its duty but more frequent the collection of low-level information
    - Data logging clarification: trace-data are very low-level ones not for the end users
  - Saving data as a function of time significantly increases the runtime
  - It cannot be perfectly assured that the parameter value observed is traced in a clock tick-by-clock tick manner
  - On some processors dedicated trace port can be found

Categories of debug tools

- Profiler:
  - Measures the dynamical performance characteristics of the program: Runtime analysis, memory usage, function calls
  - Helps to find the weaknesses of the system (which part of the program requires more memory, energy, runtime…)
  - Examples:
    - Simplicity Studio (SiLabs): Energy Profiler
      - To follow current consumption
    - Visual DSP (Analog Devices DSPs):
      - Runtime of functions in percent
      - How tasks changes, what is their time dynamic



Statistical Profiling: BLACKFIN Memory 1

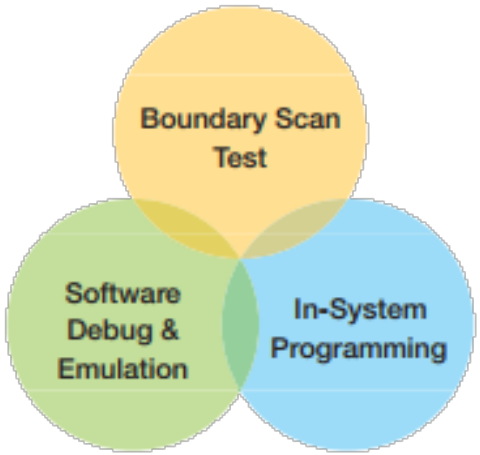
Histogram	%	Execution Unit
	92.68%	main()
	6.11%	Median_Filter()
	1.21%	Sport0_RX_ISR()

Embedded debug possibilities

- For a full debugging service, built-in (embedded) debug peripheral is needed
- Debug peripheral has access to the internal processor core, its peripherals, registers, memory
- Some typical debug interface:
  - JTAG
  - SWD: Serial Wire Debug
    - Debug interface for ARM processors
    - It can be considered as a JTAG interface requiring less wires
  - Nexus

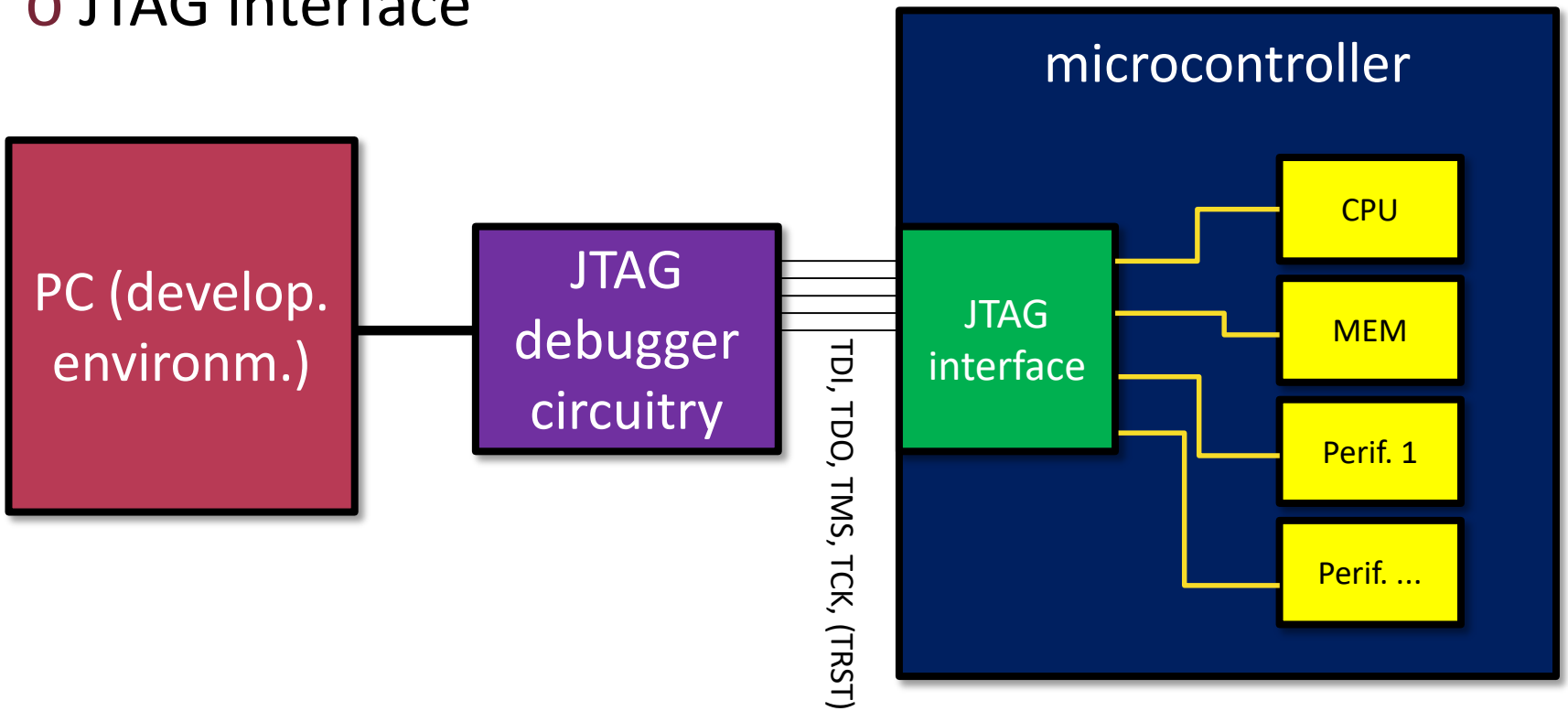
Operation of JTAG

- JTAG: Joint Test Action Group
- Original goal was: testing the electrical connections on a printed circuit board (PCB), nowadays its application is much wider
- Typical JTAG tasks:
  - Testing connection on a PCB (rarely done during development)
  - In-system programming (the circuitry not necessary to be removed during programming)
  - Debugging
- Can be connected in series
- Five wires:
  - TDI: Test Data In
  - TDO: Test Data Out
  - TCK: Test Clock
  - TMS: Test Mode Select
  - (TRST: Test Reset, optional)



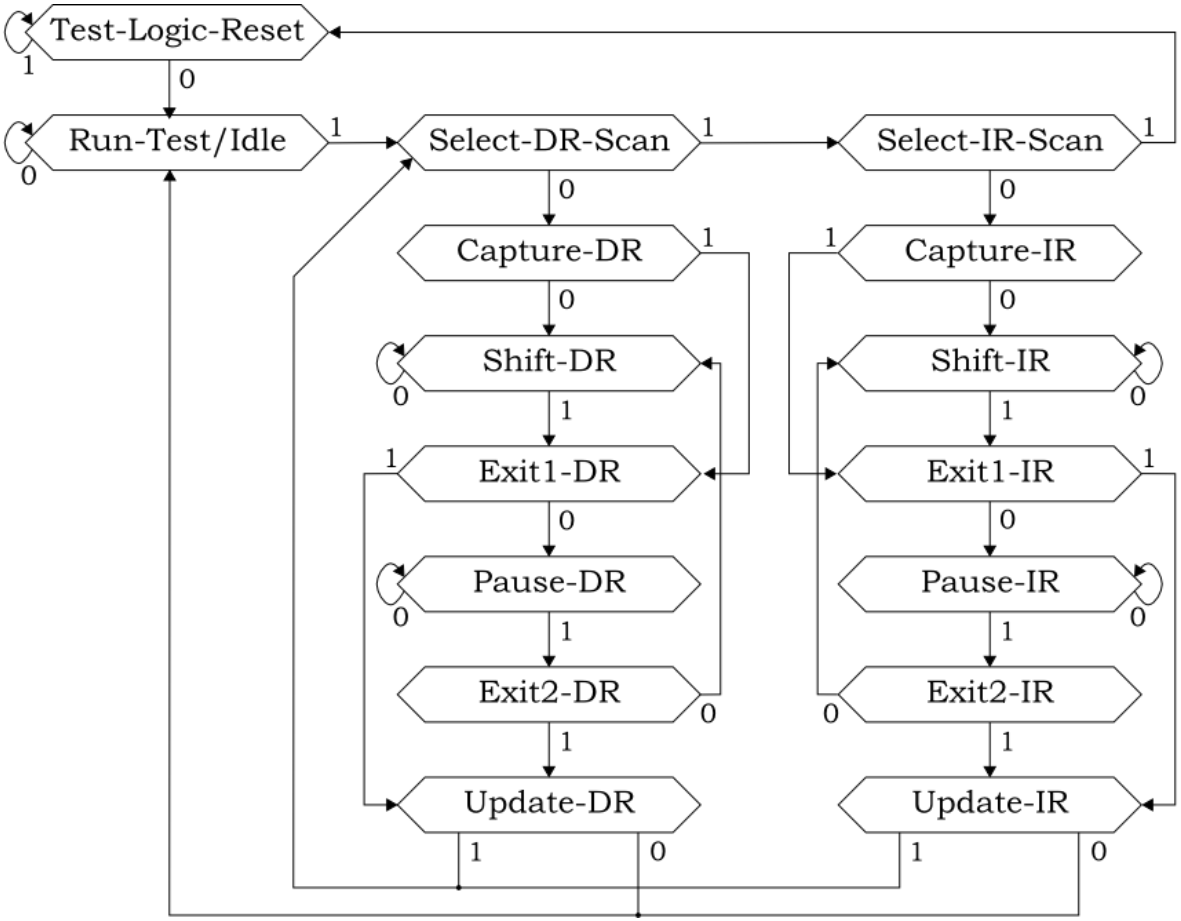
Operation of JTAG

- Architecture of the debug system:
  - PC-based development encironment
  - JTAG debugger circuitry
  - JTAG interface



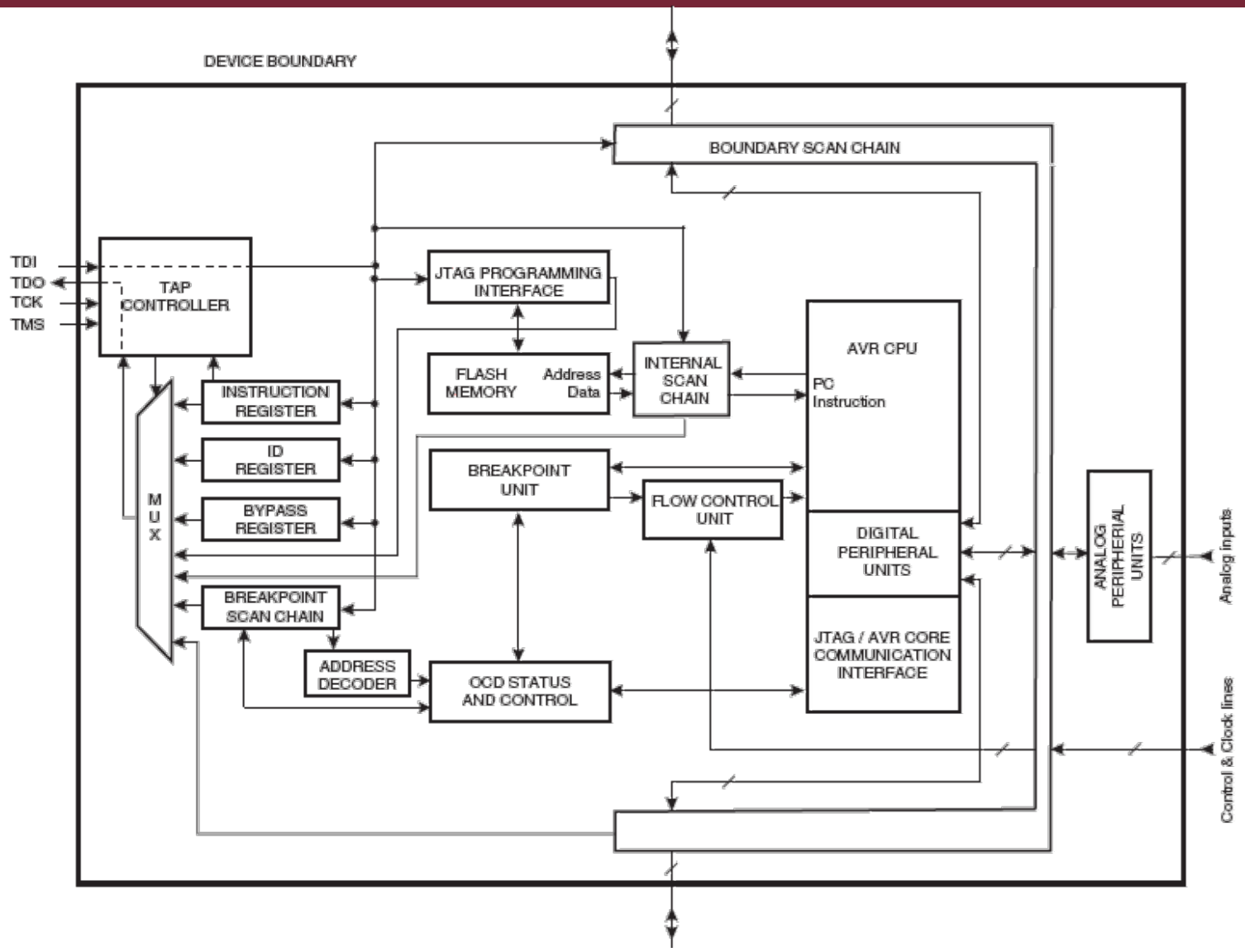
Operation of JTAG

- Structure of control: operation is controlled by a state machine
- TMS signal is used to switch between states
- Instructions can be given and data can be written in and displayed



JTAG state machine (just for illustration purposes)

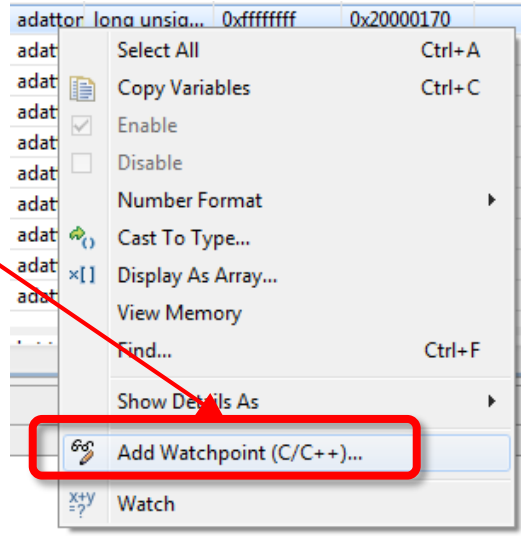
Operation of JTAG



JTAG modul block diagram: ATmega128 (just for illustration purposes)

Conditional examinations

- Breakpoint: Stopping the program running at a certain point
- Watchpoint: Stopping the program running when certain conditions are met. Examples:
  - Data watchpoint: access of data at a certain address (read and/or write)
  - Instruction watchpoint: running stops when certain instruction comes
  - HW-based support is possible, but take care since checking a condition can consume processor time
  - Example: Simplicity Studio (program stop can be set when a certain variable is accessed)



Basic debug possibilities

- In lack of dedicated debug interface own debug possibilities have to be used
- Typical questions:
  - What is the value of a certain variable?
  - Has a certain condition met?
  - Has a certain task been successfully executed?
- Typical debug possibilities:
  - Saving results into memory, reading memory (if it is supported, for programming a certain level of memory access is needed)
  - Display at GPIO pins (using LEDs or measured by oscilloscope)
    - Very limited debug possibility
    - Can be used as an indicator whether the program run into a certain condition or not, whether a code part has been executed or not
  - Sending data using a communication interface (typically UART)
  - Using a display (e.g. LCD or seven segment display)

Formatted displaying of variables

- Function *printf* puts the string on a standard output (screen), but in embedded systems no such a standard output
- Convenient: generating a formatted string, data types are well handled (e.g. printing float numbers)
- Output has to be re-routed to a certain peripheral
  - UART
  - Display
- Re-routing:
  - Compiler, more than one option is possible
  - Examples:
    - Simplicity Studio: overdefinition of _write function. This is a weak function that can be redefined. Function *printf* calls this function when starts.
    - AVR compiler: a new stream has to be defined where the pointer, that points to the function used to print to the output, has to be given. Stdout has to be re-routed to the new output

```
int _write(int file, char *data, int len) {
  SegmentLCD_Write(data);
  return len;
}
```

```
static FILE stdoutLCD  = FDEV_SETUP_STREAM(LCD_write, NULL, _FDEV_SETUP_WRITE);
  stdout = &stdoutLCD;
```

Formatted displaying of variables

- Be careful, function *printf* is resource hungry!
- Example: printing to the LCD
  - Using a normal LCD handling function
  - Program memory need: 11272 byte

```
SegmentLCD_Init(false);
SegmentLCD_Write("Pelda");
```

```
Running size tool
arm-none-eabi-size "print_proba.axf"
   text    data     bss     dec     hex filename
  11272     124      68   11464    2cc8 print_proba.axf
```

  - Printing using *printf* function by re-routing the output to the LCD
  - Program memory need : 14924 byte
  - Difference: 3652 byte, this would consume a smaller microcontroller totally up!!!

```
SegmentLCD_Init(false);
//SegmentLCD_Write("LCD");
printf("Print \n");
```

```
int _write(int file, char *data, int len) {
    SegmentLCD_Write(data);
    return len;
}
```

```
Running size tool
arm-none-eabi-size "print_proba.axf"
   text    data     bss     dec     hex filename
  14924     124      84   15132    3b1c print_proba.axf
```

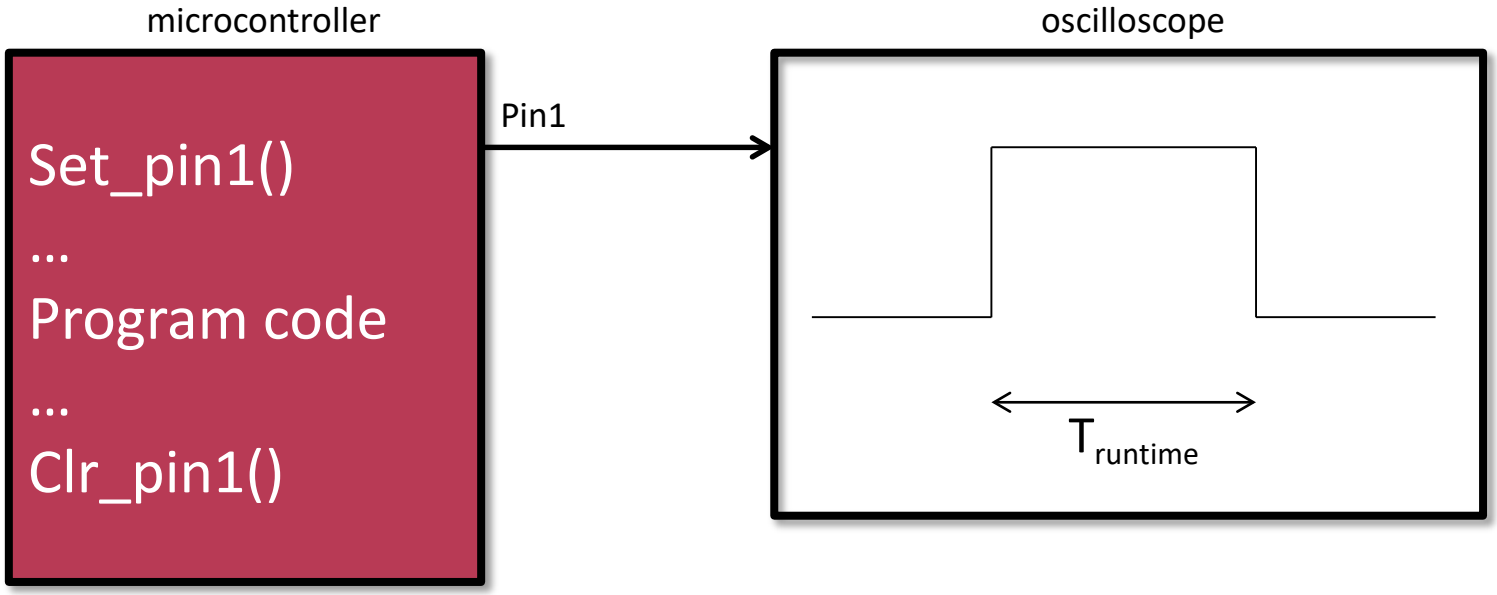
  - In case of conversion between format types runtime does matter

Measurement of runtime

- Goal of runtime measurement:
  - Checking operation (does everything run?, does the processor operate?)
  - Checking runtime requirement in terms of the time needed for the execution of the task
- Measurement of runtime:
  - Using GPIO
  - Timer
    - Special timer: Coreclock timer, that counts the CLK ticks of the processor (shows exactly how many CLK cycles has occurred since turning the device on)
  - Functions offered by the development environment

Measurement of runtime using GPIO pins

- Setting the GPIO pin into high logical level at the start of the code part whose runtime is intended to be measured and into low logical level at the end of the code part to be measured
- Requirements:
  - Available GPIO pin
  - Oscilloscope
  - Program: no special demand, try to achieve minimal overhead



Measurement of runtime using timer

- Options:
  - Starting the timer at the beginning of the code part to be measured and stopping the timer when the code part is finished
  - Starting the timer (even independently of the code part to be measured) and reading its value at the beginning of the code part to be measured then reading the timer value again when the code part to be measured reaches its end. The runtime is the time difference between the two timer values.
- Error: time needed to read timer value increases the runtime
- Core timer: special timer, measures the processor runtime in CLK ticks
- Example for using the core timer:
  - ARM cortex M3 (reading and calculating the difference requires 10 CLK cycles!!!)

```
printStart = DWT->CYCCNT;
printTime = DWT->CYCCNT - printStart;
```

(x)= printStart	uint32_t	0xd7cda8
(x)= printTime	uint32_t	10 (Decimal)

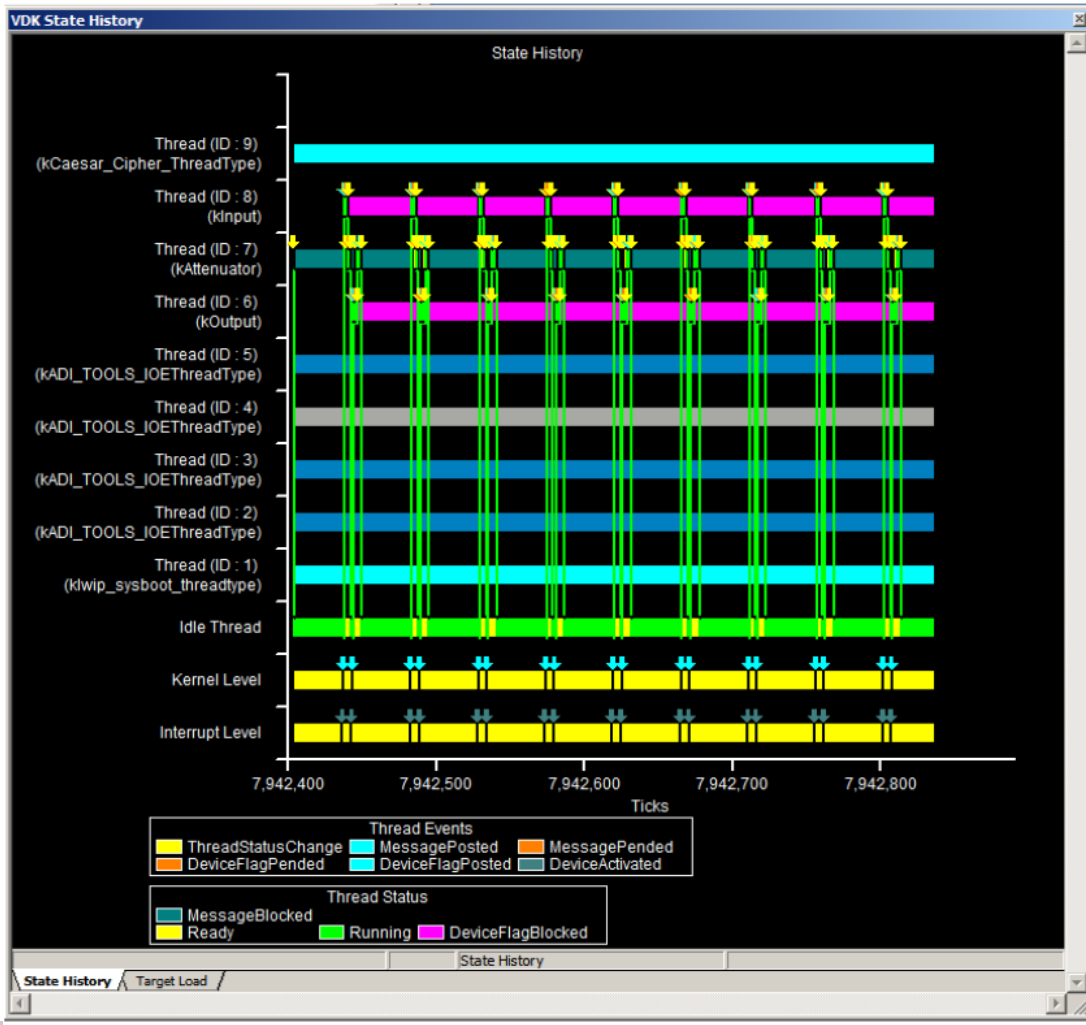
```
#define CYCLE_COUNT_START( cntr ) \
    asm("r0 = emuclk; %0 = r0;": \
    "=k" (cntr):"d" (cntr): \
    "r0")

#define CYCLE_COUNT_STOP( cntr ) \
    asm("r0 = emuclk; r1 = %1; r2 = 4; r0 = r0 - r2; r0 = r0 - r1; %0 = r0;" : \
    "=k" (cntr) : \
    "d" (cntr) : "r0", "r1")
```

Development environment (IDE)

- Runtime measurement features are available in the IDE
- Examples: see earlier at the profiler

Statistical Profiling: BLACKFIN Memory 1

Histogram	%	Execution Unit
	92.68%	main()
	6.11%	Median_Filter()
	1.21%	Sport0_RX_ISR()



Measurement of runtime

- Runtime measurement of optimized code requires extra care:
  - Markers inserted into the code for runtime measurement purposes (timer, GPIO, …) may be removed or relocated by the compiler during optimization
    - asm volatile(„nop;”); usually helps: this code is not removed by the compiler, the preceding and succeeding code parts becomes separated
  - If not only the entire code but only some code parts are tested, it may happen that during optimization the compiler removes functions if their results are not used anywhere in the rest of the code