ELEC-H410
REAL-TIME SYSTEMS

Chapter 8

Debugging the code

Debuggers

CONTENTS

- **assembly (i.e. instruction) level debug**
- debug and HLL
- debug problems

Debug in ASM

a very useful tool

- utility
  - instruction level=ASM code seldom runs at first trial
  - if user interface is limited, you just know that it does not work, but not why
  - lot of "side effects"
- type of debug commands
  - µP
  - files
  - memory
  - I/O
  - execution control
  - utilities

All debug tools offers a first low level, working on the basis of instruction cycles of the machine. This means that we can have a very close view on how the code is executed with a resolution equal to each machine instruction.
"Low level" means here "close to the processor", but also "highly detailed".

This level is very useful:
- for code written is assembly language, which is prone to problems since you can make fatal mistakes when you write the source code
- if you have problems with code written in a high-level language and want to see what is the actual machine code, which helps to detects errors like for instance casting errors

Debug in ASM

register commands

- display register contents
  - GPR: parameters, local variables
  - SFR
    - PC: monitor execution
    - Status Register
    - SP: Stack problems
- modify registers
  - parameter of a function
  - *register* variables
  - index of a loop
  - test conditional instructions

To control the microprocessor, we must above all know the state of the main registers:

- PC (Program Counter) to monitor the execution
- SP (Stack Pointer) to monitor the stack (overflow, PUSH/POP missing or in wrong order)
- PSW (Program Status Word) to be aware of events like carry and overflows in arithmetical operations
- GPR (General Purpose Registers) to know their value (global variables, local variables, parameters, loop indexes, ...)

The modification of the registers during the execution is very useful
- to force the value of a parameter passed to a function
- to accelerate the end of a loop by forcing the index to a value close to the limit
- to test the conditional instructions by forcing conditions

Debug in ASM

I/O commands

- files
  - load code in program memory
  - dump memory area (data or code) => file
- I/O ports
  - µC : ports on external pins
  - µP : special memory space for peripherals

The machine code produced by the software development chain resides in an output file. This file has to be loaded into the program memory of the system.

Conversely, the content of a portion of memory (e.g. data acquired on the system and placed in a memory buffer) can be dumped on a file.

Some commands are also related to the Input/Output space of the processor. Here we must make a difference between microcontrollers and microprocessors:
- in microcontrollers, I/O ports and I/O peripherals can be directly accessed by registers
- in some microprocessors there are two different address spaces using the same address bus, but different control signals
  - Memory read/MemoryWrite to access memory space
  - IORead/IOWrite to access I/O space

Therefore, debuggers for microprocessors must provide additional command for I/O access.

Debug in ASM

memory commands

- dump a zone (HEX et ASCII)

```
13DF:3000  65 75 72 20 64 27 82 63-72 69 74 75 72 65 20 73   eur d'écriture s
13DF:3010  75 72 20 70 82 72 69 70-68 82 72 69 71 75 65 0D   ur périphérique.
13DF:3020  0A 02 25 31 02 25 31 02-25 31 02 25 31 01 09 0A   ..%1.%1.%1.%1...
13DF:3030  20 3C 52 45 50 3E 20 20-20 20 03 08 20 08 02 0D    <REP>    .. ...
```

- ASM directly in memory
- disassembly

```
13DF:1002 56            PUSH    SI
13DF:1003 E8AA00        CALL    10B0
13DF:1006 5E            POP     SI
```

- modify a byte or a group of bytes
- search for a pattern
- fill a block (e.g. to test RAM)
- move a block
- compare blocks

Debug in ASM

execution commands /!\ not real-time

- **start/stop** execution
  - keyboard or mouse action
- **breakpoints**: stop execution if a condition is TRUE
  - mainly based on address
  - can be much more sophisticated
    - access to a variable
    - value of a variable
    - after n interations of a loop
    - .....
  - **/!\ not real-time**
- **trace**: record and display execution

Debug in ASM

trace the execution /!\ not real-time

- execution with display of
  - disassemby of the code (machine code in ASM)
  - "watch windows" for registers or variables
- variants
  - only a portion of the code
  - only branch instructions
- speed considerably reduced **/!\ not real-time**
  - execution speed = display speed
  - **step-by-step**
    - instruction by intruction
    - **step into**: follow all CALL and BRANCH
    - **step over**: do not trace CALLed functions (e.g. sinus)

Tracing the program means **executing** it and **simultaneously display**
- disassembled code, i.e. the machine code translated in assembly language
- registers
- chosen data fields (variables, stack, heap,..)

Continuous tracing generates a considerable quantity of information, so that tracing s generally restricted to the portion of the code which you want to debug.

The scrolling of the information on the screen is obviously too fast: the human eye cannot detect problems on-the-flight; one of the most popular modes of tracing is thus **step-by-step** by pressing a key or clicking the mouse. To variants are used

- ***step into*** in which all the instructions are displayed including the function calls
- ***step over*** in which function calls are skipped (functions are executed, but their execution is not traced); this mode is particularly useful to avoid entering into functions that you do not want to debug because you did not write them
  - run-time functions to manipulate strings or do mathematical computations (log, sin, ...)
  - calls to the OS

It is sometimes possible to trace **only branch and call** instructions to monitor the gross course of the execution.

The main drawback of tracing is its **total incompatibility with the real-time**, because this operation considerably slows down the execution.

Debug in ASM

most used: breakpoints

- principle
  - normal execution until breakpoint reached
  - when stopped: observe/modify the state an
- variants
  - counters (break on n^{th} occurrence)
- mecanisms
  - small µP: replace op-code at the breakpoint by SWI
  - modern µP : special break registers generate a *trap*
    - when Program Counter = breakvalue
    - Read and/or Write access to a variable
    - only method for µP with cache or pipeline

We are often interested by the behaviour of a program in the neighbourhood of a particular instruction.

In this case we can place a **breakpoint** on this instruction. The execution is normal until the breakpoint is reached and the program is stopped. The debugger will the display the current state and allow all desired modifications.

To debug **loops**, it is interesting to be able to stop the execution at the **n^{th} passage** by the breakpoint.

Two different mechanisms are used to install breakpoints depending on the processor:

- for the simple processors, the debugger replaces the opcode instruction at the breakpoint by a **software interrupt** (SWI) whose interruption vector points towards a function inside the debugger.

- for the advanced processors (from the 80386 in INTEL x86 family), the debugger can configure special registers to specify the break condition; when this one is met, an exception (trap) occurs and the corresponding interrupt vector gives the control to the debugger. The break condition can be much richer
  - on an opcode
  - when reading/writing at a particular address

Breakpoint registers are the only solution for the complex processors including caches and pipelines, because only the control unit of the processor is aware of what happens inside it.

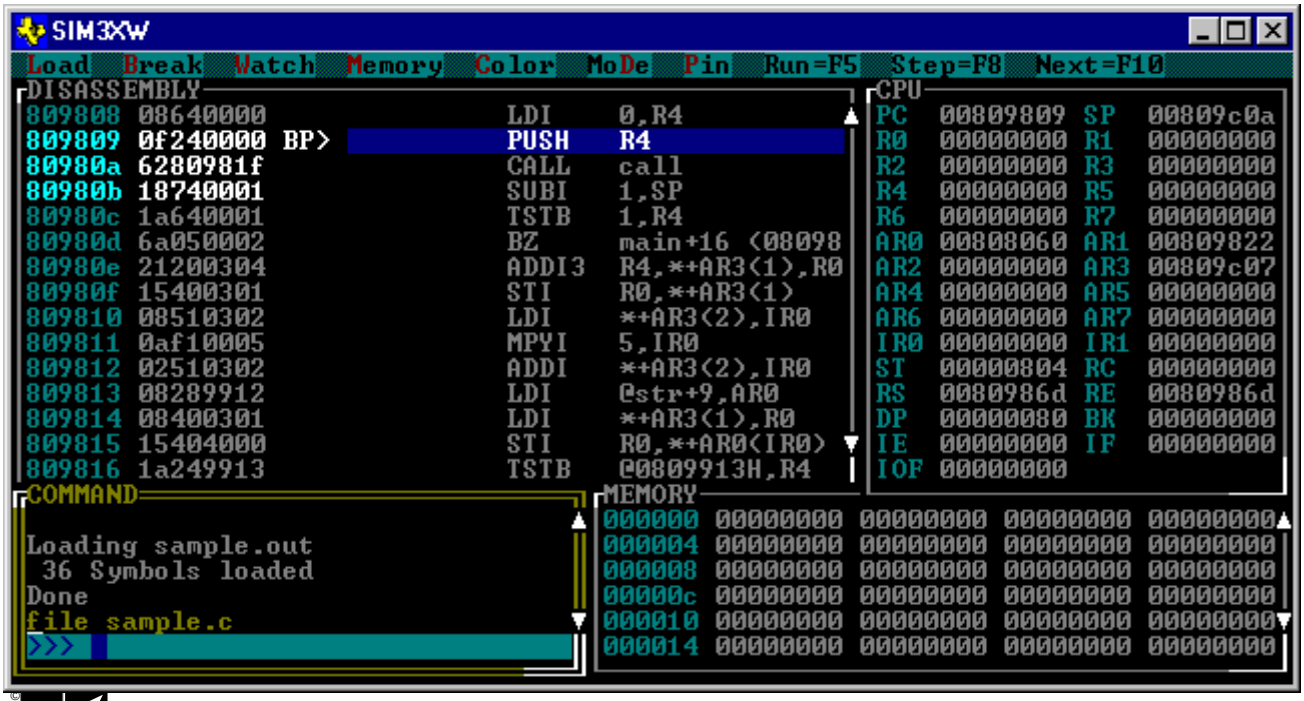
Debug in ASM

drawbacks of *breakpoints*

- **not real-time**
  - program is stopped at *breakpoint*
  - program is slowed-down to increment pass-counters for breakpoints
- simple processors: SWI is limited
  - replacing opcode by SWI is only possible in RAM => no debug for code in ROM
  - break only on opcodes
    - cannot break every time you access to a variable
    - variables can only be watched when you break

Debug in ASM

exemple of screen



A typical screen for low-level debug comprises typically several windows.

- **disassembly** of a portion of the code
- **CPU** with a view of all registers, which can be modified when the processor is stopped
- **command** for entering debug commands
- **memory** to view the content of a portion of memory

Debuggers

CONTENTS

- debug in ASM
- **debug and HLL**
- debug problems

Debug & HLL

specificities of HLL

- ASM programmers make low-level critical errors
  - alter contents of main registers
  - unpaired PUSH/POP
  - forget RTI
- HLL compilers do(should) not, but the programmer still makes other types of mistakes in
  - complex data structures
  - recursivity
  - pointers /!\
  - dynamic variables (heap or stack overflow)
  - typecasting
  - OOP

We have seen that it is advised to work in high-level language to increase the productivity, and to facilitate legibility and maintenance. It is still necessary to use debug tools, even if the compiler does not make the fatal errors that can happen when we write in assembly language and, in particular, when we manipulate directly crucial registers.

In fact, working in high-level language generates other problems at a higher hierarchical level like:

- the handling of complex data structures (tables, structured variables....)
- recursivity
- pointers
- allocation of dynamic variables and monitoring of *heap*/*stack* errors and overflows
- typecasting errors ocurring when the compiler converts a variable to another type
- errors related to object oriented programming (OOP)

Debug & HLL

HLL debugger need extended power

- ASM debuggers not adapted to HLL
  - machine code <=> ASM
  - machine code <=> HLL
    - ? translation of IF, CASE, FOR ...
    - ? translation of variables and functions names

HLL listing has no direct relation with ASM debug screen => tiresome intellectual effort

- solutions
  - poor man's debugging
  - listing ASM or pseudo-asm + XREF
  - HLL debugger

Debug & HLL

poor man's debugging

- activate compiler run-time checks (=> traps)
  - index overflow
  - math errors (DIV 0, overflow) (often intern to CPU)
  - memory allocation, heap/stack collision
  - type conflicts
- **signs of activity**
  - **on screen messages (/!\ slow)**
  - **messages on serial I/O, network**
  - **bits toggle on port I/O, LED, ...**
- OS error messages
  - illegal operations (access to protected memory)
  - error in files
  - default handlers for CPU traps

Debug & HLL

drawbacks of the *run-time*

- more code
- slower execution (☹ if small margin)
- **standard handling of exceptions** by compiler run-time or OS is **not sufficient**
  - offending task is often preempted and killed
  - in which state is the system ? is it safe ?
- **your responsability**
  - develop **adapted error handling** functions
  - **black box** (cyclic buffer recording state continuously)
  - enable processor **watchdog and manage "hot reset"**
  - **redundancy** (software, hardware, as simple as possible)

Debug & HLL

use mixed listing and ASM debugger

```
; testled.c    7  unsigned char i;
; testled.c    8  for(i=0;i<10;i++)
  ?LINE 8
  MOV _4,#00H
  _3 :
; testled.c    9  {
; testled.c   10  P1_0=0;
  ?LINE 10
  CLR 090H
; testled.c   11  delay_ms(500);
  ?LINE 11
  MOV R7,#0F4H
  MOV R6,#01H
  LCALL _?delay_ms
; testled.c   12  P1_0=1;
  ?LINE 12
  SETB 090H
; testled.c   13 delay_ms(500);
  ?LINE 13
  MOV R7,#0F4H
  MOV R6,#01H
  LCALL _?delay_ms
  ?LINE 8
  INC _4
  MOV A,_4
  ADD A,#0F6H
  JNC _3
; testled.c   14  }
```

loop index

param = (R6,R7)

loop index

- compile in ASM mode
  - each HLL line => ASM instructions
- HLL => ASM
  - see compiler manual
    - coding of different types
      char, int, float, double,...
    - parameter passing
      order, registers, pointers
    - how do functions return their value
    - conventions for names
      modules, variables, functions

This figure illustrates one of the means to combine writing in HLL and debugging in assembly language. You can activate an option of the compiler producing a mixed listing, where each line of C appears as a comment in the assembler listing.

It is certainly a help but this method requires a very good knowledge of the documentation of the compiler in particular to know:

- how are different types coded?
  char, int, float, double,...
- how are passed the parameters to a function?
  - in which order?
  - in which registers in non-reentrant mode?
  - based on which pointer in reentrant mode?
- how does a function return its value?
- which are conventions of names between the C and the assembler for modules, variables and functions?
- .....

Looking at this listing, we can for example deduce that:

- the function delay_ms is written in non-reentrant mode and that the integer parameter is passed via the two 8 bit registers R6&R7 with the most significant byte in R6 (indeed $500_{10}=1F4_{16}$)
- the index i of the loop is a direct access variable with label _4
- that the end of loop is obtained by overflow of the subtraction (i-10) ($F6_{16}=10_{10}$)

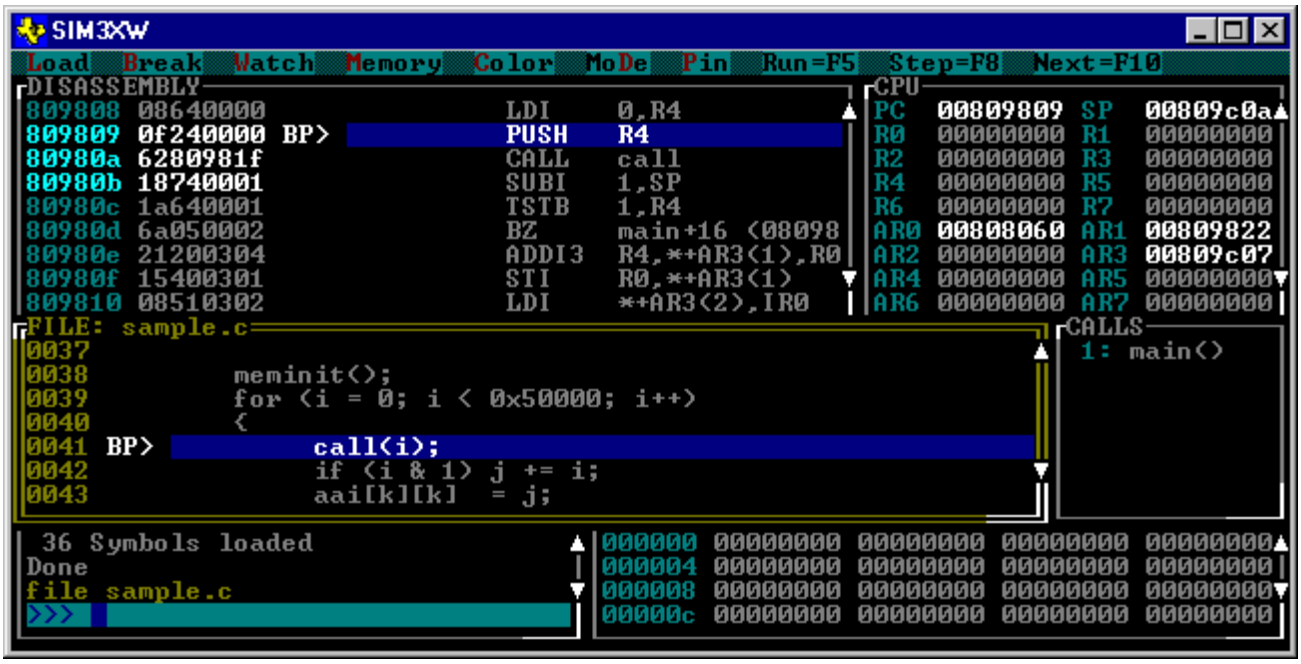
Debug & HLL

use mixed listing and ASM debugger: avoid it !

- drawbacks
  - effort of documentation
  - voluminous listing => just do it for small pieces of code
  - very tiresome operations
    - monitor function: parameters (stack)
    - monitor function: local variables (stack)
    - monitor dynamic variables (stack)
    - going inside structured variables
    - multiple precision arithmetics
    - encoding of floating point numbers
    - typical exercise: compute the value of a "double" parameter passed to a function via the stack

Debug & HLL

typical screen



The best solution is obviously to use a specific debugger for the high-level language. This figure shows us an example of a screen, which is to be compared with the previous example of an assembly language debugger. They use the same processor and belong to the same development environment.

Compared to the other example, we see the appearance of two additional windows: one for the C source code, the other one for the *call stack*, i.e. the chain of successive function calls if they are nested (here only the function main() is active).

The breakpoint (BP) is visible at the same time in C and in assembler.

Debug & HLL

HLL debugger = ASM debugger plus

- *breakpoints* on
  - # line of source code
    - the compiler has to generate a symbol for each line !
  - function calls
  - modification of a variable (watch window)
    - ! possible only during
      - trace / step
      - function calls
    - ! extra code required (option of the compiler)

Debug & HLL

HLL debugger = ASM debugger plus

- *trace / step*
  - per processor instruction (same as ASM)
  - per source code line
    - *step into* functions to trace the internals
    - *step over* functions just to get the result
  - only function calls
  - *traceback* or *call stack* (you are in function x(), called by y(), called by main())
- variables
  - monitor / modify with same syntax as source code(ex: 1.27e-6, "low battery")
  - monitor parameters and local variables of a function during its execution

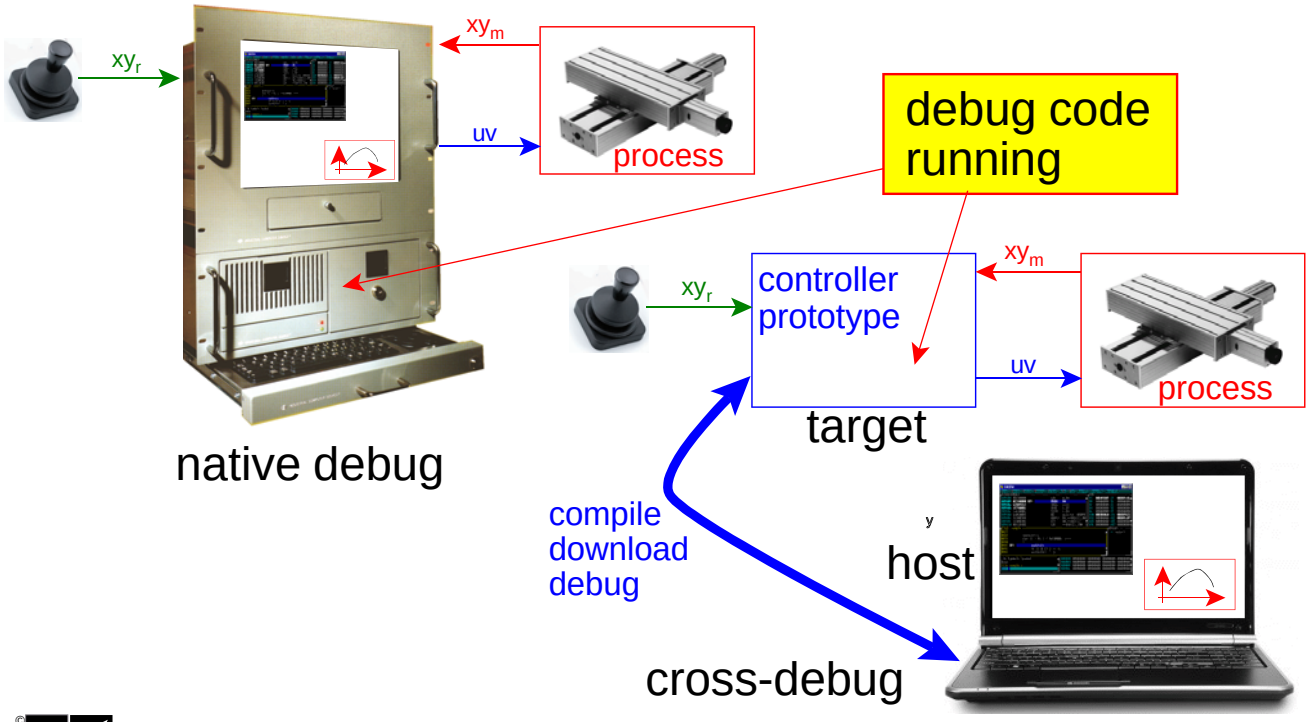
Debuggers

CONTENTS

- debug in ASM
- debug and HLL
- **debug problems**

Problems of the debuggers

structures of debug



Developing and debugging the code directly on the target is called **native development**. It generally happens when the target is a complete microcomputer like the industrial PC on the left figure.

In most embedded system, the software can also be developed on a **host** PC and downloaded to the **target** platform, for several reasons
- the target platform runs on a CPU which is not powerful enough to make a suitable development platform
- the target runs on an RTOS and the development tools do not run on this OS

If the developer has to install on the host PC a development environment for another processor, all tools get the prefix "cross"
- cross-assembler
- cross-compiler
- cross-linker
- cross-debugger

This is called **cross-development.**

In any case, the debugging process requires that some additional debug code will have to run concurrently with the program.

Problems of the debuggers

concurrence with your program

- some debugger code always run on the target machine
  - needs memory (can be a problem for small systems)
  - needs CPU time
  - if SWI are reserved for the debugger, you cannot use them in the program
  - a bug in the program can generate
    - the destruction of the debugger in memory
    - infinite loops
    - /!\ beware of critical sections: only NMI still works
    - reset => you lose all information (except black box)
  - best case: **multi-tasking well-protected OS**
    - tasks and OS are shielded by the memory management unit (virtual memory) a task can kill neither the debugger, nor the OS
    - OS can take control back (time-slicing, key pressed, ...)
    - **not the case for "tiny" OS and many µC have no MMU**

We usually make native development, i.e. on the machine where the program will be executed (the typical case is the PC). Hence the debugger and the program coexist in memory and execute concurrently. This coexistence is not always simple.

- in systems with severe limitation of the memory, it is not always possible to release the place necessary to the execution of the debugger

- the debugger uses software interrupts that can no longer be used in the programs

- a severe hanging of the program (like a stack overflow or infinite loop) can lead, in weakly protected systems, to the crushing of the memory in which the debugger resides; the detection of the errors becomes quite difficult in this case

- hanging the program in a critical section is a major accident, no event can give the control back to the debugger except an NMI or a watchdog interrupt

- if the only solution is to press RESET, the state of system is lost, and it becomes very difficult to fix this bug (the only way is to trace the execution by writing regularly traces of the execution
  - in a memory area that will not be cleaned at reset
  - on I/O ports, where they will be recorded by a logic analyser

The **debuggers will work best in a well-protected multitasking environment** . The combination of a processor equipped with a memory management unit (MMU) and an advanced operating system (NT and successors, Linux, VRTX, QNX....) makes it possible to shield the OS and the debugger against faults in the programs.

Unfortunately most microcontrollers do not have an MMU and run tiny OSes; hence the protection level is rather low.

Problems of the debuggers

not compatible with real-time

- real-time is impossible
  - trace
  - step
  - breakpoint
- real-time more difficult
  - slow down due to CPU overhead
    - run-time checks
    - writing on the activation stack
    - monitoring of variables
    - suppression of optimization
  - easier with an RTOS
    - if CPU is not overloaded by debugger => debug is schedulable

It is not easy to respect the real-time constraints during debugging.

The first reason is the fundamental incompatibility between tracing (and a fortiori step-by-step) and the concept of real-time.

During the development phase, the code is often bulkier and slower because the compiler options generate more code:
- run-time checks are activated to track errors
- creation of an activation stack on which information is written each time a function is called or returns to the caller
- instrumentation of the code for the monitoring of certain variables and events
- impossibility (sometimes) to activate the optimization and debugging options of the compiler, because debugging requires a more systematic code.

The resort to an RTOS offers additional possibilities to instrument the code provided there is enough spare CPU time for the debugging task, so that the set of tasks is still schedulable. You will have the occasion to see it in more in details at the laboratory.

Problems of the debuggers

- volume of the code is increased
- interrupt service routines (**ISR**)
  - difficult to trace
  - sometimes impossible to place breakpoints
  - if the system is based on periodic interrupts, trace or step breaks the frequency (real-time violation)
  - **debug ISR carefully before using them**
- mesuring execution time is difficult
  - easier in simulation for small µC
  - use external pin and scope or *logic analyser* (see)
- debug of code in ROM requires an *emulator*

CONCLUSIONS (1)

- debuggers
  - are extremely useful
  - are intrusive
    - in CPU time
    - in volume of code
    - in potential conflicts
- do not exempt you to write your code properly; on the contrary, the more you code rigorously, the more effective your debug sessions will be.

CONCLUSIONS (2)

good practice rules

- develop progressive, modular code
  - write small test programs for each function
  - test all combinations of parameters
  - keep those test program for the future (maintenance)
- debug ISR carefully and apart
- think safety and security
  - FMECA (Failure Modes Evaluation and Critical Analysis)
    - find every single point of failure
    - eliminate it
    - add redundancy
      - special techniques of software
      - hardware: low-level, as simple as possible
  - implement a black-box
    - on-line diagnosis (restart or not after a watch-dog reset)
    - post-mortem diagnosis

CONCLUSIONS (3)

good practice rules: instrumentation of the code

- external signs of activity
  - power-on self tests
  - enter/exit a function
  - publish value of some variables
  - status of tasks (ready, running, preempted, suspended,...)
- how ?
  - add messages on communication port
    - RS-232, RS-485, network
    - easier with an RTOS
      - add communication task
      - if schedulable, you are in real-time
    - create a log vector/file on the host
  - bits on GPIO visible on oscilloscope or logic analyser
    - 8 bits=256 messages !
    - extremely fast

target has to be slightly overdimensioned

CONCLUSIONS (4)

good practice rules: be able to interact

- target should accept real-time commands
  - change parameter of controller (Kp, Ki)
  - change "verbosity" of your program
    - (de)activate messages
      - in a portion of the code
    - for a certain "level"
- how
  - communication port
  - response time 0.1 s
  - you should get an acknoledgement