Embedded Programming

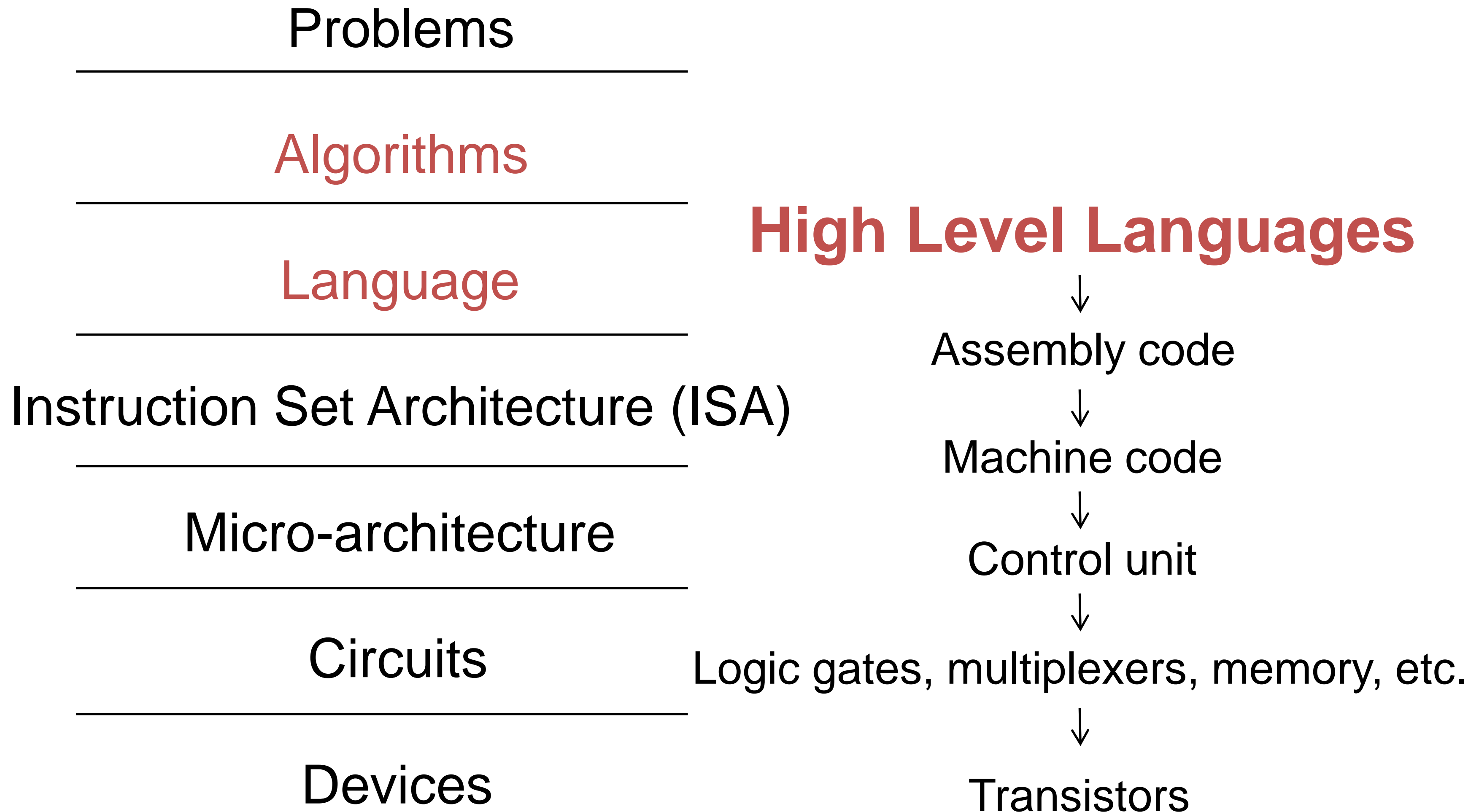
Giacomo Paci
Domenico Balsamo
Micrel Lab – DEIS
giacomo.paci@unibo.it domenico.balsamo2@unibo.it

Outline

Lesson:

- High Level Languages
- Compilers
- Programming and IDE (Integrated Development Environment)
- Application Programming Interface (API)
- Memory Mapping and Segmentation
- Embedded Debugging
- Writing the Code
- Variables and Declarations
- Data Types
- Operators
- Tips
- Examples

Levels of Abstraction



High Level Languages

High level programming languages provide **abstraction** from the underlying hardware and Instruction Set Architecture (ISA)

- Hide low level details (ISA) from programmer
- Uniform interface (not tied to ISA) to program
- Portable software (works on different ISAs)
- The compiler generates the machine code

Allow us to use symbolic names for values

- Programmer simply assigns each value a name and can ignore *many* memory details, such as:
  - register usage
  - variable allocation
  - loads and stores
  - stack management for subroutine calls

Advantages:

- Easier to write and read
- Easier to debug
- Easier to maintain

The C Programming Language

- The C programming language is perhaps the **most popular programming language** for programming embedded systems
- Few embedded systems have capability for dynamic linking, so if **standard library functions** are to be available at all, they often need to be directly linked into the executable as **static libraries**
- C remains a very popular language for micro-controller developers due to the **code efficiency** and **reduced overhead and development time**.
- C offers **low-level control** and is considered **more readable than assembly**
- Additionally, using **C increases portability**, since C code can be compiled for different types of processors
- Many **free C compilers** are available for a wide variety of **development platforms**. The compilers are part of an Integrated IDEs with In-Circuits Emulators (ICE) support, breakpoints, single-stepping and an assembly window

Compilation vs. Interpretation

Interpretation: An *interpreter* reads the program and performs the operations in the program

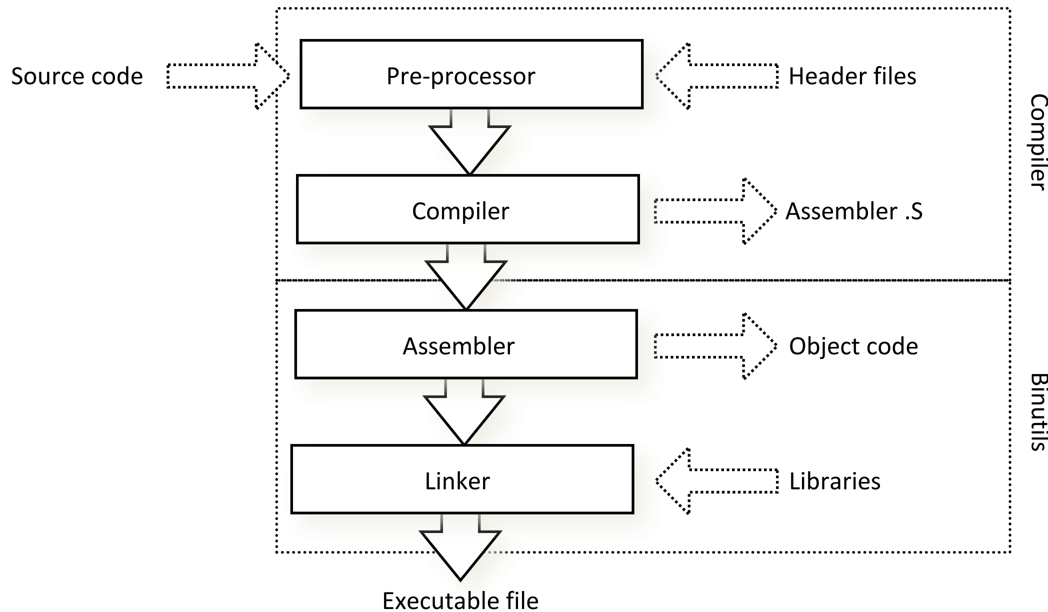
- The program **is not executed directly**, but is executed by the interpreter.
- Every instruction is decoded into machine code and executed.

Compilation: A *compiler* translates the program into a machine language program called *executable image*.

- The *executable image* is **directly executed** to the hardware.
- The whole program is transformed into machine code at once, optimizing the program flow.

Embedded systems use compiled programs which are directly executable.

Compiler



- **Preprocessor**: The preprocessor handles directives for source file inclusion (`#include`), macro definitions (`#define`), and conditional inclusion (`#if`)
- **Compiler**: transforms source code into assembler modules
- **Assembler**: creates object code by translating assembly instruction mnemonics into opcodes
- **Linker**: it is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

Crosscompiler

A **cross compiler** is a compiler capable to create executable code for **a platform other than one in which the compiler is running**. Cross compiler tools are used to generate executable code for embedded system or multiple platforms.

- The **GNU Compiler Collection** (usually shortened to **GCC**) is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU toolchain
- It has been ported to more kinds of processors and operating systems than any other compiler
- Cross compiling GCC requires that a portion of the target platform's C standard library is available on the host platform
- **Sourcery G++** is a complete development environment with C/C++ compilers and run-time libraries, a source- and assembly-level debugger, an IDE, and many other tools designed with the embedded developer in mind

Integrated Development Environment

An integrated development environment (**IDE**) also known as integrated design environment or integrated debugging environment is a software application that provides comprehensive facilities to computer programmers for software development

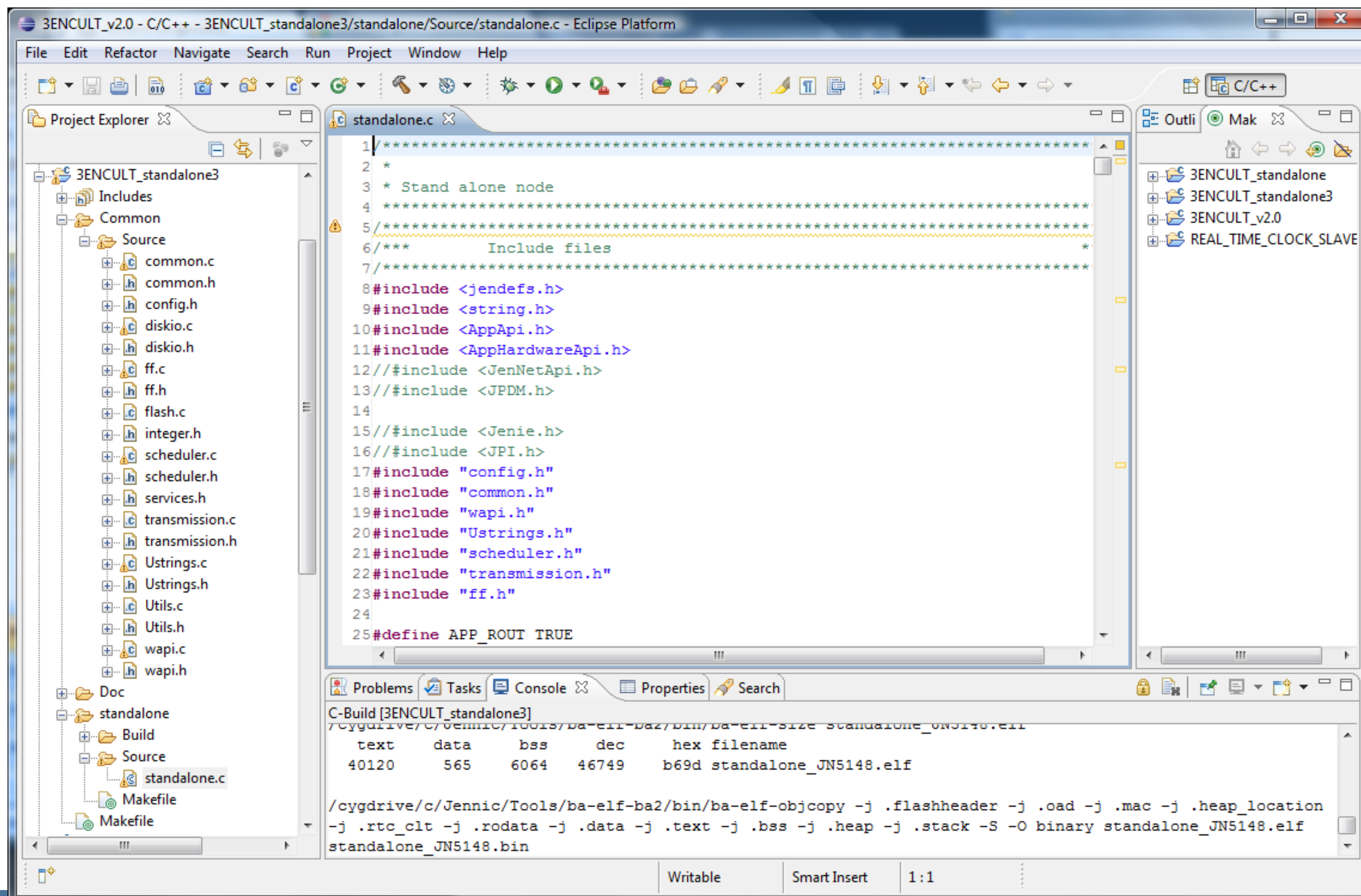
An IDE normally consists of:

- source code **editor**
- **compiler** (or cross-compiler)
- **Assembler**
- **Linker**
- **build automation tools** (compiler, assembler and linker manager)
- **debugger**

IDEs are designed to **maximize programmer productivity** by providing tightly-knit components in a unique user interface, avoiding the programmer to switch to several programs.

Embedded IDE examples

- Commercial: IAR, Ride7, AVR Studio, etc...
- Powerful open source IDEs based on **Eclipse** (+ Plugins + auto-configuration packets)
- **Eclipse** is a open source IDE that supports several microarchitectures (e.g. ARM, INTEL, STM32, JN5148 etc.)



Make file

- It is a file with all information and parameters for the compiler and linker to make the executable program.

Project files

The compiler needs the list of files that has to compile

```
82 #SDK_BASE_DIR          = $(abspath
83 #APP_BASE                  = $(absp
84 #APP_BLD_DIR           = $(APP_BASE
85 #APP_SRC_DIR           = $(APP_BASE
86
87 ####################################
88 # Application Source files
89
90 # Note: Path to source file is found
91 APPSRC += ${TARGET}.c
92 #APPSRC += Utils.c
93 APPSRC += wapi.c
94 APPSRC += Ustrings.c
95 APPSRC += common.c
96 APPSRC += scheduler.c
97 APPSRC += transmission.c
98 APPSRC += flash.c
99 APPSRC += ff.c
100 APPSRC += diskio.c
101
102 ##########################################################
103 # Additional Application Source directories
104 # Define any additional application directories outside th
105 # e.g. for AppQueueApi
106
```

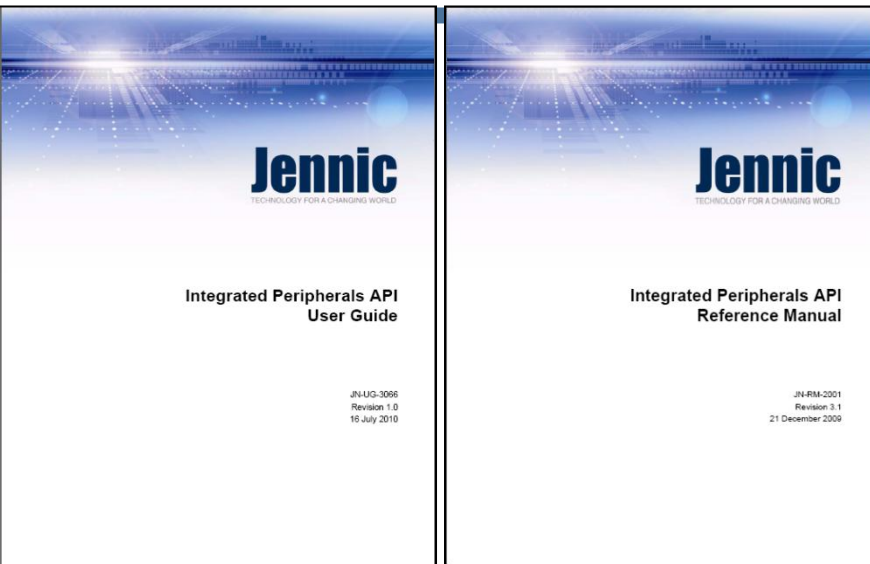
Project Explorer: Includes; Common; Source: common.c, common.h, config.h, diskio.c, diskio.h, ff.c, ff.h, flash.c, integer.h, scheduler.c, scheduler.h, services.h, transmission.c, transmission.h, Ustrings.c, Ustrings.h, Utils.c, Utils.h, wapi.c, wapi.h; Doc

Know the executable program can be generated only pressing BUILD button

C and Application Programming Interface (API)

- Modern **optimizing compilers** are claimed to compile high-level languages into code that can run **as fast as hand-written assembly**
- Since every microcontroller has its own architecture and structure each vendor provides libraries to interact with hardware, as peripheral devices.
- **Application Programming Interface** (API) are functions provided by the vendors or open source communities to use the microcontroller’s devices.
- The **API**s are stored in libraries that can be provided as: C code, binary code; and can be located either in the IDE or in the microcontrollers ROM.
- There are **API**s even for complex firmware structure as: Radio Stack (IEEE802.15.4, ZigBee, Bluetooth, WiFi); Data bus (USB, Konnex, CANBUS, RS232, RS485, Ethernet) and data storage (SD card, file system).
- Usually there are **API**s for manage **interrupts**.

APIs Example (JN5148)



- Jennic provides a **user guide** manual and a **reference** manual for its integrated peripherals API.

- On the right there are the APIs to interact with the digital Input/output lines.

Jennic — Integrated Peripherals API Reference Manual

5. DIOs

This chapter describes the functions that can be used to control the digital input/output lines, referred to as DIOs. The Jennic wireless microcontrollers have 21 DIO lines, numbered 0 to 20, where each DIO can be individually configured. However, the pins for the DIO lines are shared with other peripherals (see list below) and are not available when those peripherals are enabled:

- UARTs
- Timers
- Serial Interface (2-wire)
- Serial Peripheral Interface
- Intelligent Peripheral Interface
- Antenna Diversity

For details of the shared pins, refer to the datasheet for your wireless microcontroller.

In addition to normal operation, when configured as inputs, the DIOs can be used to generate interrupts and wake the device from sleep.

The functions are listed below, along with their page references:

Function	Page
vAHI_DioSetDirection	88
vAHI_DioSetOutput	89
u32AHI_DioReadInput	90
vAHI_DioSetPullup	91
vAHI_DioSetByte (JN5148 Only)	92
u8AHI_DioReadByte (JN5148 Only)	93
vAHI_DioInterruptEnable	94
vAHI_DioInterruptEdge	95
u32AHI_DioInterruptStatus	96
vAHI_DioWakeEnable	97
vAHI_DioWakeEdge	98
u32AHI_DioWakeStatus	99

APIs Example (JN5148)

vAHI_DioSetDirection

```
void vAHI_DioSetDirection(uint32 u32Inputs,
                          uint32 u32Outputs);
```

Description

This function sets the direction for the DIO pins individually as either input or output (note that they are set as inputs, by default, on reset). This is done through two bitmaps for inputs and outputs, *u32Inputs* and *u32Outputs* respectively. In these values, each of bits 0 to 20 represents a DIO pin, where bit 0 represents DIO0 and bit 20 represents DIO20 (bits 21-31 are ignored). Setting a bit in one of these bitmaps configures the corresponding DIO as an input or output, depending on the bitmap.

Note that:

- Not all DIO pins must be defined (in other words, *u32Inputs* logical ORed with *u32Outputs* does not need to produce all zeros for bits 0-20).
- Any DIO pins that are not defined by a call to this function (the relevant bits being cleared in both bitmaps) will be left in their previous states.
- If a bit is set in both *u32Inputs* and *u32Outputs*, it will default to becoming an input.
- If a DIO is assigned to another peripheral which is enabled, this function call will not immediately affect the relevant pin. However, the DIO setting specified by this function will take effect if/when the relevant peripheral is subsequently disabled.
- This function does not change the DIO pull-up status - this must be done separately using **vAHI_DioSetPullup()**.

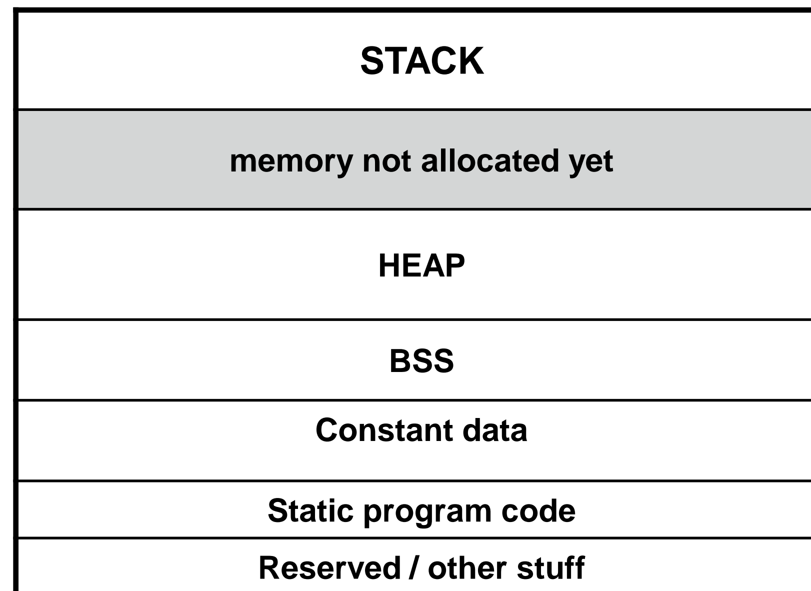
Parameters

u32Inputs	Bitmap of inputs - a bit set means that the corresponding DIO pin will become an input
u32Outputs	Bitmap of outputs - a bit set means that the corresponding DIO pin will become an output

Memory Segmentation

In a computer system using **segmentation**, an instruction operand that refers to a memory location includes a value that identifies a segment and an offset within that segment. A segment has a **set of permissions, and a length**, associated with it

The code **allocates a number of special purpose memory blocks for different data types**



Segments

In classical architectures there are five basic memory areas:

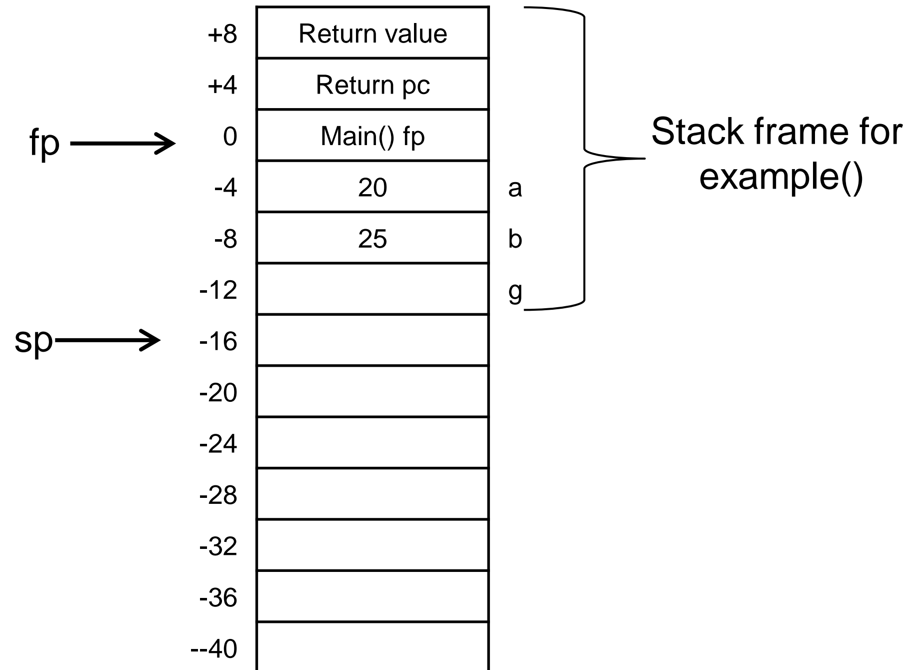
- **DATA**. The Data area contains **initialized global and static variables.**
- **BSS**. The BSS area contains **uninitialized global and static variables**.
- **HEAP**. The Heap area is usually managed by **operative system** and it is shared from different treads, tasks, functions etc. dynamically. It is not suggested to use heap area without OS, otherwise the programmer has to guarantee the not data overlapping or collision.
- **STACK**. The stack is a **LIFO** structure, typically located in the higher parts of memory. It usually "grows down" with every write. It is used to manage the call and the local variable of the functions.
- **CODE**. Also known as a text segment or simply as text, is a memory area used to contain **executable instructions**. It has a static size and is usually read-only.

Stack example

```
void main(void){
        int a;
        a=example();
}

int example(void){
        int a = 20, b = 25;
        int g;                    <- pc
        g = func(72,73);
        g=g+a;
}

int func(int x, int y){
        int a;
        a=y-x;
        return a;
```



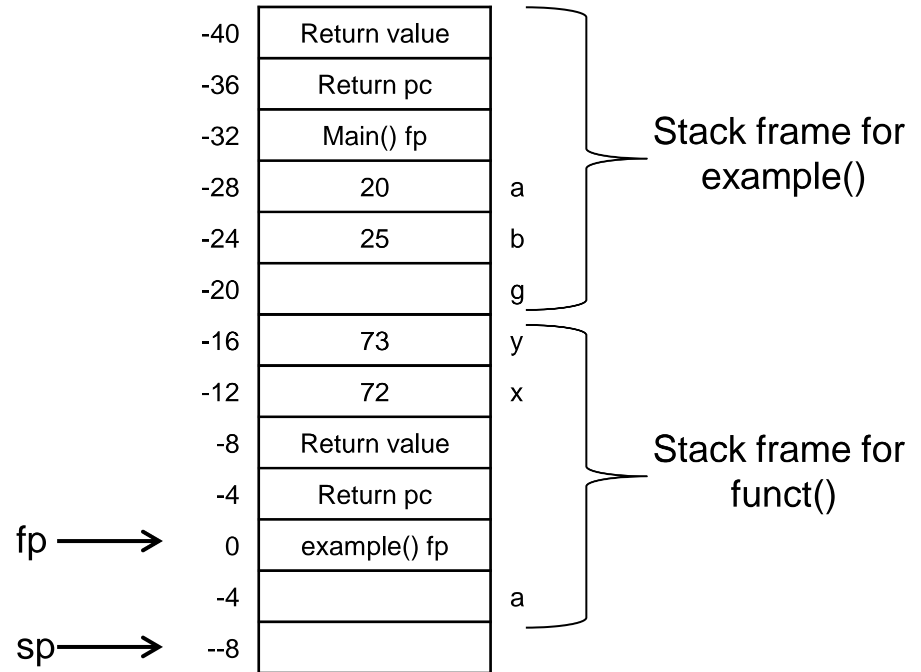
- pc – program counter: point to the current executing instruction
- fp – frame pointer: point to the start of the function frame
- sp- stack pointer: point to the next free stack space

Stack example

```
void main(void){
        int a;
        a=example();
}

int example(void){
        int a = 20, b = 25;
        int g;
        g = func(72,73);
        g=g+a;
}

int func(int x, int y){
        int a;        <-- pc
        a=y-x;
        return a;
```

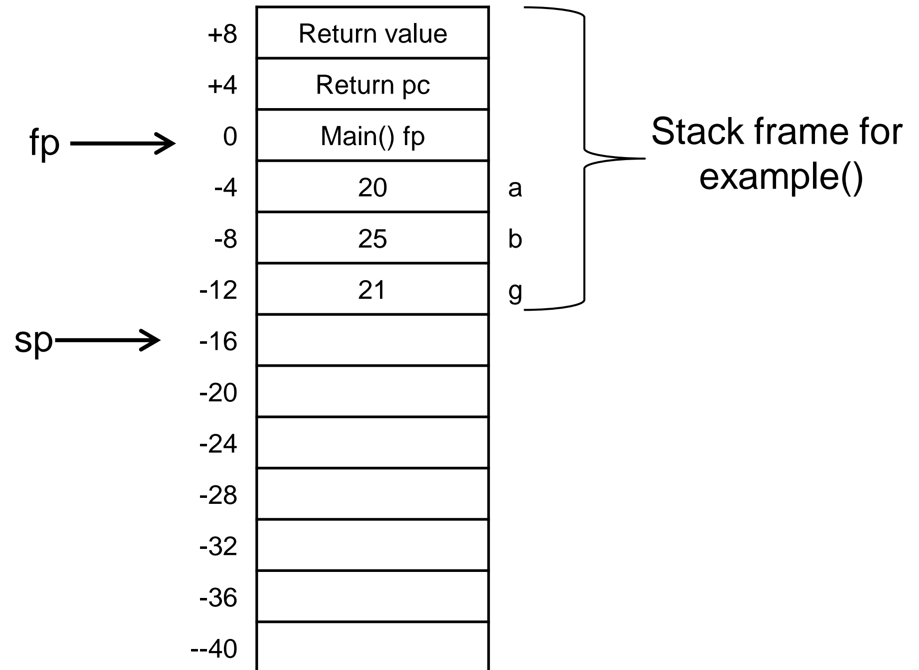


Stack example

```
void main(void){
        int a;
        a=example();
}


int example(void){
        int a = 20, b = 25;
        int g;
        g = func(72,73);
        g=g+a;              <-- pc
}

int func(int x, int y){
        int a;
        a=y-x;
        return a;
```



BSS and DATA example

```
int data_bss;          <--- data saved in the BSS

int data_data=32;      <--- data saved in the DATA


void main(void){
        int a;
        static int b;      <--- data saved in the BSS
        static int c=11;   <--- data saved in the DATA
        a=example();
}
```

```
Problems  Tasks  Console
C-Build [3ENCULT_standalone3]
Generating binary ...
/cygdrive/c/Jennic/Tools/ba-elf-ba2/bin/ba-elf-size standalone_JN5148.elf
   text    data     bss     dec     hex filename
  40120     565    6064   46749    b69d standalone_JN5148.elf

/cygdrive/c/Jennic/Tools/ba-elf-ba2/bin/ba-elf-objcopy -j .flashheader -j .oad -j .mac -j .heap_location
-j .rtc_clt -j .rodata -j .data -j .text -j .bss -j .heap -j .stack -S -O binary standalone_JN5148.elf
```

BSS area size 6064Byte

DATA area size 565Byte

Interrupts

Interrupt is an asynchronous signal indicating the need for attention or a synchronous event in software indicating the need for a change in execution.

hardware interrupt causes the processor to save its state of execution and starts the execution of an interrupt handler.

There are:

- Non-maskable interrupt (**NMI**)
- Maskable interrupt (**IRQ**)

Interrupts can be managed by:

- Single interrupt handler (The microprocessor calls the same procedure for every interrupt and the programmer has to detect which peripherals has generated the interrupt)
- Interrupt vector (The microprocessor calls a specific handler for different kind of interrupt)

Interrupt example

```
void main(void){
  int a;
  initialize_the_system;
  while(1){
    sleep();
  }
}


void interrupt_handler(void){
  printf("pressed button\n")
  toggle_led();
  return;
}
```

- The main initialize system and setup an interrupt that will be triggered when a button is pressed, after that the entire system go to sleep.
- When the user press the button the interrupt management turn on the system, putting in execution the interrupt handler that toggle a led.

There are interrupt capable to turn on the system from sleep state (e.g. GPIO interrupt, wake timer interrupt, comparator interrupt)

Interrupts are used even to reduce the power consumption permitting the CPU to go in sleep state and wake-up only for specific event

Interrupt example

```
233//manage system interrupt
234PUBLIC void sys_interrupt(uint32 u32DeviceId, uint32 u32ItemBitmap){
235    bool_t nocase=TRUE;
236    if(u32ItemBitmap & E_AHI_SYSCTRL_VREM_MASK){
237        db_printf(APP_COM_START,"enter brown out\r\n",NULL);
238        nocase=FALSE;
239    }
240    if(u32ItemBitmap & E_AHI_SYSCTRL_VFEM_MASK){
241        db_printf(APP_COM_START,"exit brown out\r\n",NULL);
242        nocase=FALSE;
243    }
244
245    if(u32ItemBitmap & E_AHI_SYSCTRL_WK0_MASK){
246        db_printf(APP_COM_START,"wake timer0 interrupt\r\n",NULL);
247        nocase=FALSE;
248        sleep_value=0x7FFFFFFF;
249    }
250
251    if(u32ItemBitmap & E_AHI_SYSCTRL_WK1_MASK){
252        db_printf(APP_COM_START,"wake timer1 interrupt\r\n",NULL);
253        vAHI_WakeTimerEnable(E_AHI_WAKE_TIMER_1, TRUE);
254        vAHI_WakeTimerStartLarge(E_AHI_WAKE_TIMER_1, 0x7FFFFFFF);
255        wapi_RTC_refresh(0);
256        extern uint32 max_sleep_time;
257        uint64 check_sleep_value=u64AHI_WakeTimerReadLarge(E_AHI_WAKE_TI
258        if(check_sleep_value < (max_sleep_time << 15)){
259            sleep_value=check_sleep_value;
260            if(sleep_value<(3276)){ //if waketimer0 is 100msec to trigge
261                vAHI_WakeTimerEnable(E_AHI_WAKE_TIMER_0, TRUE);
262                vAHI_WakeTimerStartLarge(E_AHI_WAKE_TIMER_0, 0x7FFFFFFF);
263                sleep_value+=0x7FFFFFFF;
264            }
```


3.5 System Controller Interrupts

System Controller interrupts cover a number of on-chip peripherals that do not have their own interrupts:

- Comparators
- DIOs
- Wake Timers
- Pulse Counter (JN5148 only)
- Random Number Generator (JN5148 only)
- Brownout detector (JN5148 only)

Interrupts for these peripherals can be individually enabled using their own functions from the Integrated Peripherals API.

The handling of interrupts from these sources must be incorporated in a user-defined callback function, registered using the function **vAHI_SysCtrlRegisterCallback()**. The registered callback function is automatically invoked when an interrupt of the type E_AHI_DEVICE_SYSCTRL occurs. The exact source of the interrupt (from the peripherals listed above) can then be identified from a bitmap that is passed into the function. Note that the interrupt will be automatically cleared before the callback function is invoked.

Note: For details of the callback function prototype and the interrupt source bitmap, refer to the *Integrated Peripherals API Reference Manual (JN-RM-2001)*.

***Caution:** The registered callback function is only preserved during sleep modes in which RAM remains powered. If RAM is powered off during sleep and interrupts are required, the callback function must be re-registered before calling **u32AHI_Init()** on waking.*

Embedded System Debugging

Debugging embedded systems is **different** in many aspects from traditional application debugging, due to hardware constraints and limited resources.

This makes it inconvenient or even impossible to run a software debugger together with the debugged program on the same system

Because of these restrictions, embedded systems are usually debugged using **remote debugging**

- The debugger is running on a **host computer**, and controls the target either through **hardware**, or through a **small software** running on the target
- The developer can either passively watching the code, and possibly the data flow, or by actively stopping the target at the point of interest

On embedded systems, the **hardware itself could have errors**, like an unstable memory interface or untested system components

Debug Solutions

Logic Analyzers / Trace Hardware

- Allow the program flow to be passively monitored. Logic analyzers **monitor the target's data and address bus**, and usually generate a listing of executed instructions, possibly annotated with the data accessed.
- It's not possible to trace execution within modern cached architectures, since instructions contained inside the cache wont show up on the memory interface.

In-Circuit Emulators / ROM Emulators

- An in-circuit emulator (ICE) replaces the target micro-controller with a **special debug variant**, that includes hardware debugging facilities.
- The emulator is connected to a host computer which runs the debugger software and allows both passive and active debugging.

Debug Stub

- Debug stubs, often called "debug monitors", **run on the target system, and connect to a host computer running the debug software**.
- They require working initialization code, that sets up the target clocks, main memory, and a communication channel (unsuitable for early development stages, where initialization code has to be debugged itself).

Debug Solutions - JTAG

Integrated Debug Circuitry / On-Chip Debug

- Instead of having to replace the target's micro-controller with a special debug version, **every chip shipped contains the debug functionality**. A serial communication channel, able to operate at high clock speeds, is used to connect the debug circuitry to a host debugger, allowing low pin-count debug connections

- Processors can normally be **halted**, **single stepped**, or **let run freely**. Code **breakpoints** are supported, both for code in RAM and in ROM/flash. Data breakpoints are often available, as is bulk data download to RAM

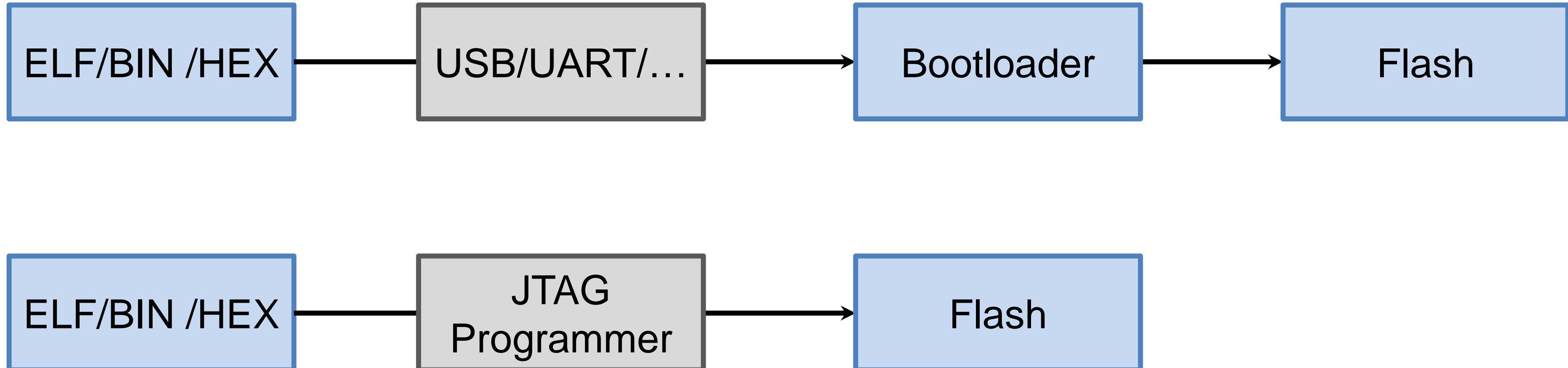
- Besides debugging, another application of JTAG is allowing **device programmer (hardware) to transfer data into internal non-volatile device memory**
  - This is usually done using data bus access like the CPU would use, and is sometimes actually handled by a CPU, but in other cases memory chips have JTAG interfaces themselves

JTAG for Programming

To **program** a device we have two alternatives:

1) Using a **USB/UART/...** connection in **bootloader** mode

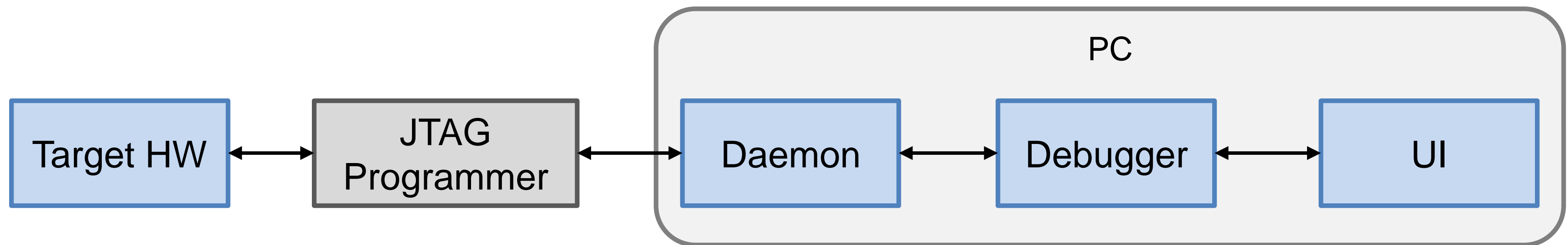
2) Using **JTAG and programmer** to write flash memory



JTAG for Debugging

We need at least 5 elements to be able to debug a micro-controller:

1) target hardware
2) JTAG programmer
3) a debugger
4) a daemon interfacing debugger and JTAG programmer
5) a user interface toward the debugger (CLI/GUI)



JTAG for Debugging: Programmer

The JTAG programmer is a **debug adapter hardware**

It generally refers to a small adapter that **attaches to your computer** via USB, Parallel Printer / Serial Port or Ethernet cable

The JTAG module provides **access to JTAG operations**. Different JTAG hardware interfaces define their own implementation of an abstract interface, and may use common code that's suitable for many different devices

JTAG for Debugging: Debugger

The GNU Debugger, usually called just **GDB** and named gdb as an executable file, is the standard debugger for the GNU software system. It is a portable debugger that runs on many Unix-like systems and works for many programming languages

The purpose of a debugger such as gdb is **to allow you to see what is going on "inside" another program while it executes**

gdb can do four main kinds of things to help you catch bugs in the act:

1) Start your program, specifying anything that might affect its behavior (not in embedded)

2) Make your program stop on specified conditions

3) Examine what has happened, when your program has stopped (memory, registers, etc...)

4) Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another

JTAG for Debugging: Daemon and CLI

The GNU Debugger (GDB) implements a remote protocol for use over serial lines that's also being used on TCP/IP network connections. The protocol specifies packets that are used by GDB to control the target's operation

It is the interface between debugger and JTAG programmer

We need an interface to "talk" with GDB, e.g. a Command Line Interface (CLI):

```
target remote 127.0.0.1:61234 break mainloadmonitor resetcontinue
```

The first instruction is for daemon connection

Writing the Code

The programming style is **very** important!

- The style used in writing the code constrains the ease with which the program can be **read, interpreted or redeployed**;
- The **maintenance of the written source code**, when adhering to a coding standard is easy to interpret, maintain and change;
- Programmers can develop their own programming style. However, for team work, they must meet a **set of rules that provide uniformity** to allow easy interpretation of the code by all team members;

Writing the Code

- Use **lots of comments** to provide enough information to fully understand the code

```
/* This is a comment */
```

- Comment each procedure telling:

```
/*----------------------------------*/
/* ProcedureName - what it does     */
/* Parameters:                      */
/*   Param1 - what param1 is        */
/*   Param2 - what param2 is        */
/* Returns:                         */
/*   What is returned, if anything  */
/*----------------------------------*/
```

- Use lots of **white space** (blank lines)

- Always use **indentation**:
  - **Align** code blocks to **highlight** the functionality
  - Each new scope is indented 2 spaces from previous
  - Put { on end of previous line, or start of next line
  - Line matching } up below

```
if(a < b)
{
  b = a;
  a = 0;
}
else
{
  a = b;
  b = 0;
}
```

Variables

Variables are **symbolic names** that hold **values**

- Variable declarations include
  - A symbolic name
  - Type (int, char, double)
  - Scope (code region where the variable is defined)
- Variables are stored in **memory** or in **registers** and the compiler keeps track of where a variable value is currently stored and when it needs to be moved from memory to a register, or from a register to memory or deleted.
- Variables definitions conventions:
  - **Data types**: start with a capital letter, e.g. `Line` or `AverageVelocity`
  - **Variables**: start with a letter and should avoid abbreviated names or the use of ambiguous name, e.g. `line`, `averageVelocity`
  - **Constants**: all in capital letters, e.g. `RED`, `GREEN`
  - **Functions**: use verbs and always start with a small letter, e.g. `calculateVelocity()` or `activateOutput()`
  - **Iterative control** operation variables are identified as `i`, `j`, `k`
  - The variables should be **initialized** when they are declared
  - The use of **global variables** should be **avoided**: they permanently use memory, but can make execution faster

Declaration: Local versus Global

Declaration of variables:

- Must always be made at the beginning of a program;
- **Global** variables:
  - Declared outside of a function
  - Accessible throughout the code
  - Stored in Global Data Section of memory
  - Scope is entire program
  - Initialized to zero
- **Local** variables:
  - Declared within one function and are only accessible during its execution
  - Declared at the beginning of a block
  - Stored on the stack
  - Scope is from point of declaration to the end of the block
  - Un-initialized
- Encapsulate your variables!
  - Avoid global variables.
  - Keep declarations as close as you can to where you use the variables– keep the scope as small as you can.
  - It’s better to explicitly pass parameters to functions, rather than use global variables to pass data.

Data Types

C data types

Type	Size [bits]	Representation	Range values	
			Minimum	Maximum
signed char	**8**	**ASCII**	**-128**	**127**
char, unsigned char	**8**	**ASCII**	**0**	**255**
short, signed short	**16**	**2’s complement**	**-32768**	**32767**
unsigned short	**16**	**Binary**	**0** ^{31}	^{31} **65535**
int, signed int	**32**	**2’s complement**	**-2**	$2^{32} - 1$
unsigned int	**32**	**Binary**	**0** ^{63}	$2^{63} - 1$
long, signed long	**64**	**2’s complement**	**-2**	$2 - ^{64} 1$
unsigned long	**64**	**Binary**	**0**	**2**

Identifiers

- ***`const`***:
  - Used to declare a constant (content is not changed in the course of code implementation);
  - Stored in program section memory.
- ***`extern`***:
  - Used to make reference to variables declared elsewhere, for example in another module.
- ***`register`***:
  - Used to store a variable in a processor's register;
  - Promotes faster access to the contents of the variable;
  - Only used locally and depends on the register's availability.
- ***`static`***:
  - Declared within a function or a program block;
  - Preserves the variable even after a function or block has been executed.
- ***`volatile`***:
  - A statement using this descriptor informs the compiler that this variable should not be optimized.

Operators

C supports a rich set of operators that allow the programmer to manipulate variables

Assignment – (=)
- changes the values of variables

Arithmetic – (+ - * / %)
- add, subtract, multiply, divide, module

Bitwise – (& | ^ ~)
- AND, OR, XOR, NOT, and shifts on Integers

Relational – (== != <)
- equality, inequality, less-than, etc.

Logical – (&& || !)
- AND, OR, NOT on Booleans

Increment/Decrement (++ --)

Order of Evaluation

Precedence

- The order in which operators and expressions are evaluated

Associativity

- The order which in which operators of the same precedence are evaluated
  - Left to Right for +, – , *, /, %
  - Right to Left for some other operators

Parentheses

- Override the evaluation rules

```
a - b / c * d + e * f
```

```
(( a - (( b / c ) * d )) + ( e * f ))
```

Good software practices for low power consumption

Principles for low power applications:

- Maximize the time in standby;
- Use interrupts to control program flow;
- Replace software functions with peripheral hardware;
- Manage the power of internal peripherals;
- Manage the power of external devices;
- Device choice can make a difference;

Effective code is a must.
Every unnecessary instruction executed is a portion of the battery wasted that will never return.

Embedded Program Structure

Typical structure of an embedded program:

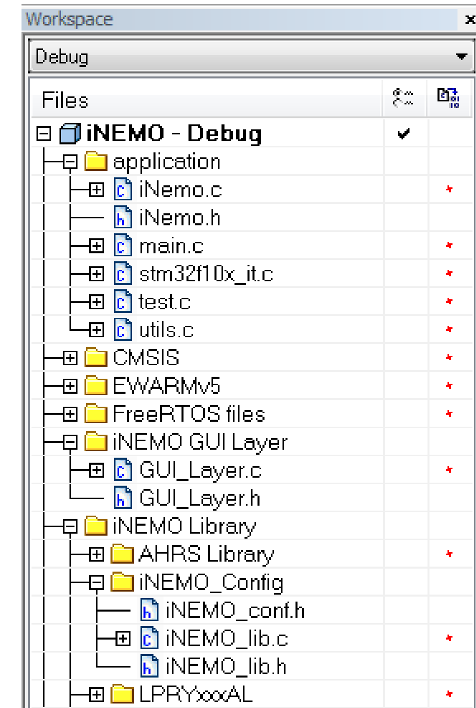
1. Uses libraries to abstract hardware – header inclusion;
2. Variable declaration and initialization;
3. Hardware initialization:
   - Only the used hardware is initialized
   - Everything else is OFF to save power
4. External hardware initialization, if present:
   - Init external devices (memories, sensors, …) present on the board
5. Set up Interrupts and timers to organize the program flow;
6. Main loop (usually loops forever):
   - Execute operations needed at every iteration and if can it goes to sleep
   - External and asynchronus events are handled via interrupts

Programming Tips

Programming Tips:

- Organize your code in several **smaller files**!
  - It is more redable and manageable
  - Use different files for different hardware parts or different program functionalities
  - Easier to **reuse code**

- Use the outputs to control the program flow:
  - LEDs to signal the different status of the device
  - `printf` to check the output results and debug



Embedded Programming

Giacomo Paci
Domenico Balsamo
Micrel Lab – DEIS
giacomo.paci@unibo.it domenico.balsamo2@unibo.it