5TC option AUD

Embedded Programming Basics

Romain Michon, Tanguy Risset

Labo CITI, INSA de Lyon, Dpt Télécom

GRAME-CNCM

6 septembre 2020

Introduction to Embedded systems

ESP32 Presentation

Embedded Peripherals Programming

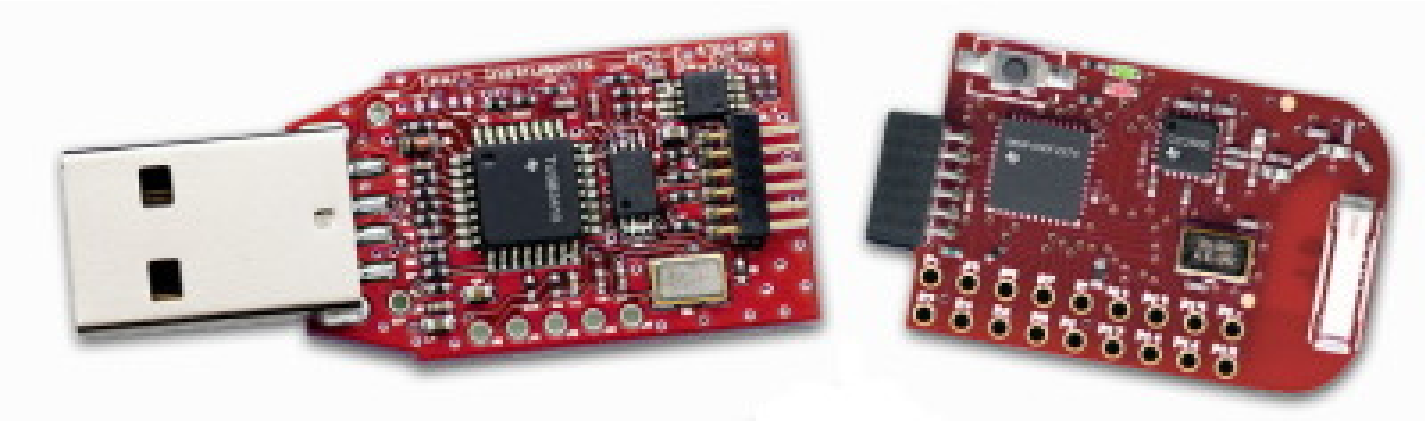
Interrupt in Embedded Programming

- Pile d'exécution
- Interruptions

Different types of embedded systems

- Micro-controler and sensor networks : (example : MSP430)
- Embedded computing devices (Sink nodes, phones, tablets, example : raspberyPi)
- Micro-controleur 32 bits with a bunch of RAM (example : ESP32)

Example of Small Embedded System



- **eZ430-RF2500** with a MSP430f2274 and a radio chip CC2500
- MSP430 ($\simeq$ 1€))
- 16 bits processor, 16Mhz maximum
- 64KB of addressable memory, 1KB RAM
- Usual micro-controler peripherals
- No MMU
- Low power design

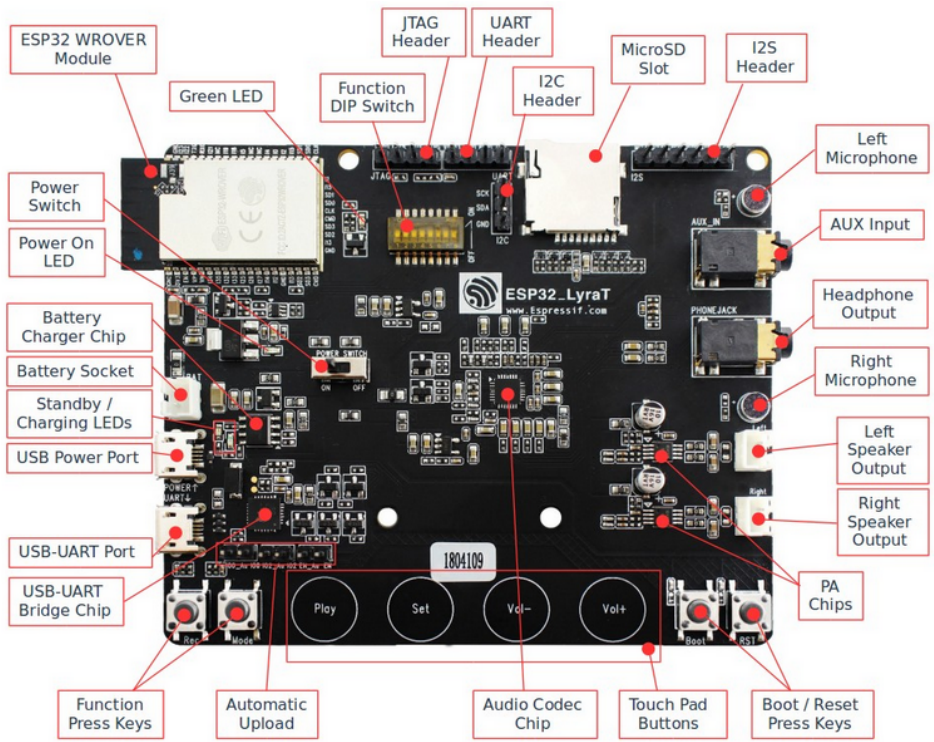
Example of More powerfull Systems

- **Beagleboard** ($\simeq$ 150€)
- Arm Cortex A8 1 GHz
- DaVinci SoC ARM+DSP
- Puce graphique 3D
- 512 MB of DDR SDRAM
- 4GB SD-Card
- DVI-D, S-Video, 4 port USB Hub, Stereo In/Out, Ethernet 10/100...

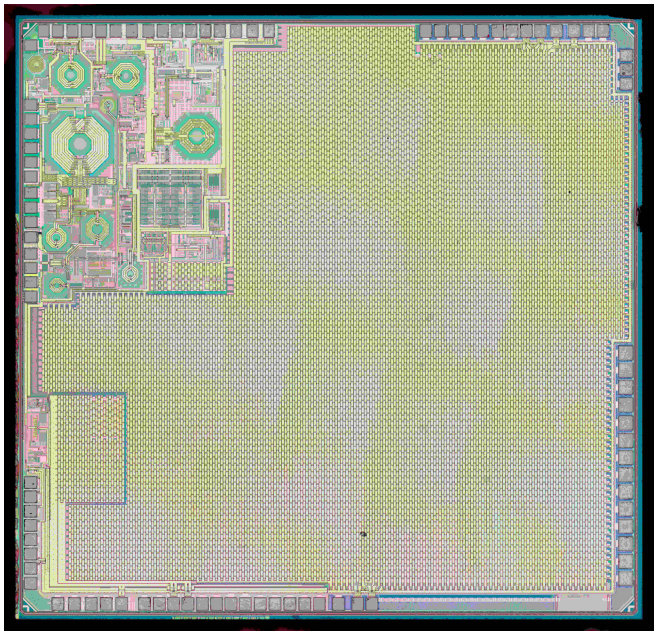
- **Rasberry Pi** ($\simeq$ 25€)
- Broadcom BCM2835, 700 MHz ARM avec FPU
- GPU Videocore 4
- RAM 512 Mo.
- 4GB SD-Card
- video RCA 2 port USB Hub, Stereo Out, Ethernet 10/100...

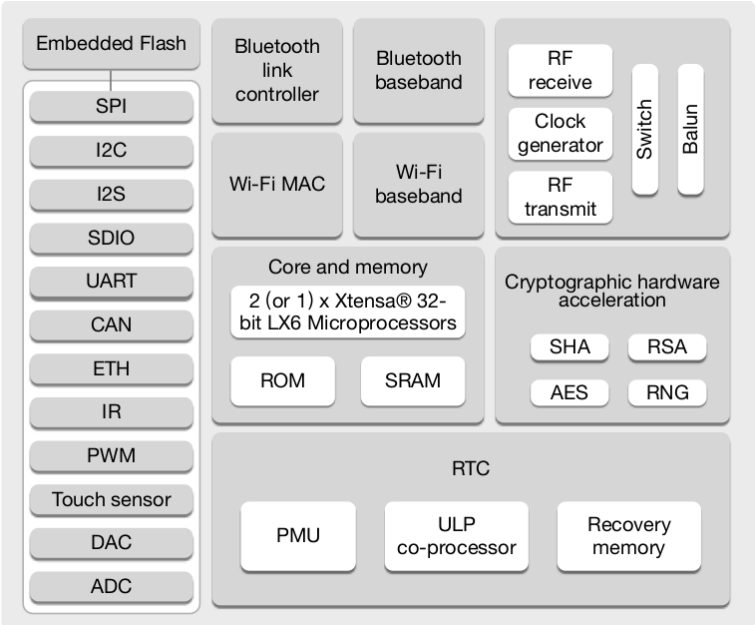
ESP32 - LyraT : intermediate system

- **ESP32** ($\simeq$ 4€)
- Xtensa dual Core (160MHz)
- 520 KB of RAM
- Audio sound card and touch sensors
- Usual micro-controler peripherals
- Wifi and Blutooth integrated chips



ESP32 presentation

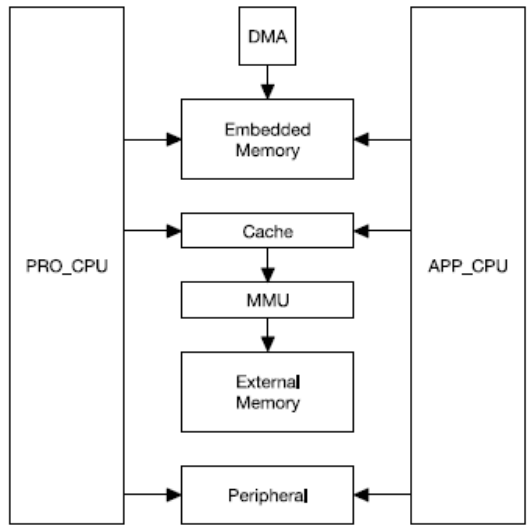




- Single 2.4 GHzWi-Fi and Bluetooth combo chip
- Xtensa Dual-core 32-bit LX6 microprocessor that can reach 600 MIPS
- 448 KB of ROM, and 520KB of SRAM
- Many peripherals...

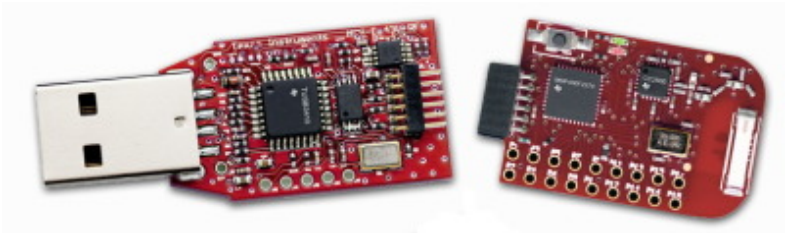
2 Cores (Quasi Symetric)

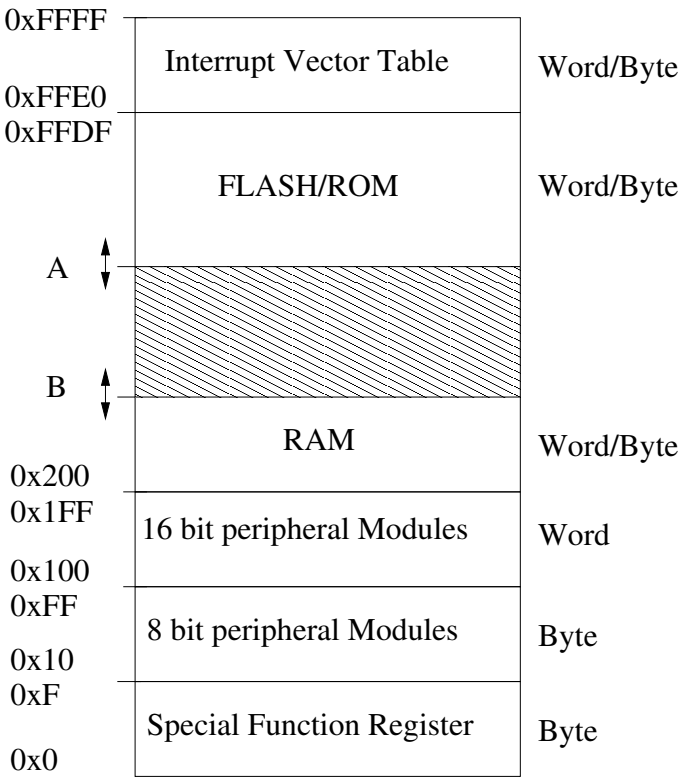
- The two CPUs are named “PRO_CPU” and “APP_CPU” (for “protocol” and “application”),
- The data bus and instruction bus are both little-endian
  - If from address 0x000000 we encounter the following bytes 0x0, 0x1, 0x2, 0x3, it represents Integer 0x03020100
- Each CPU can directly access
  - embedded memory through both the data bus and the instruction bus,
  - external memory which is mapped into the address space (via transparent caching & MMU)
  - peripherals



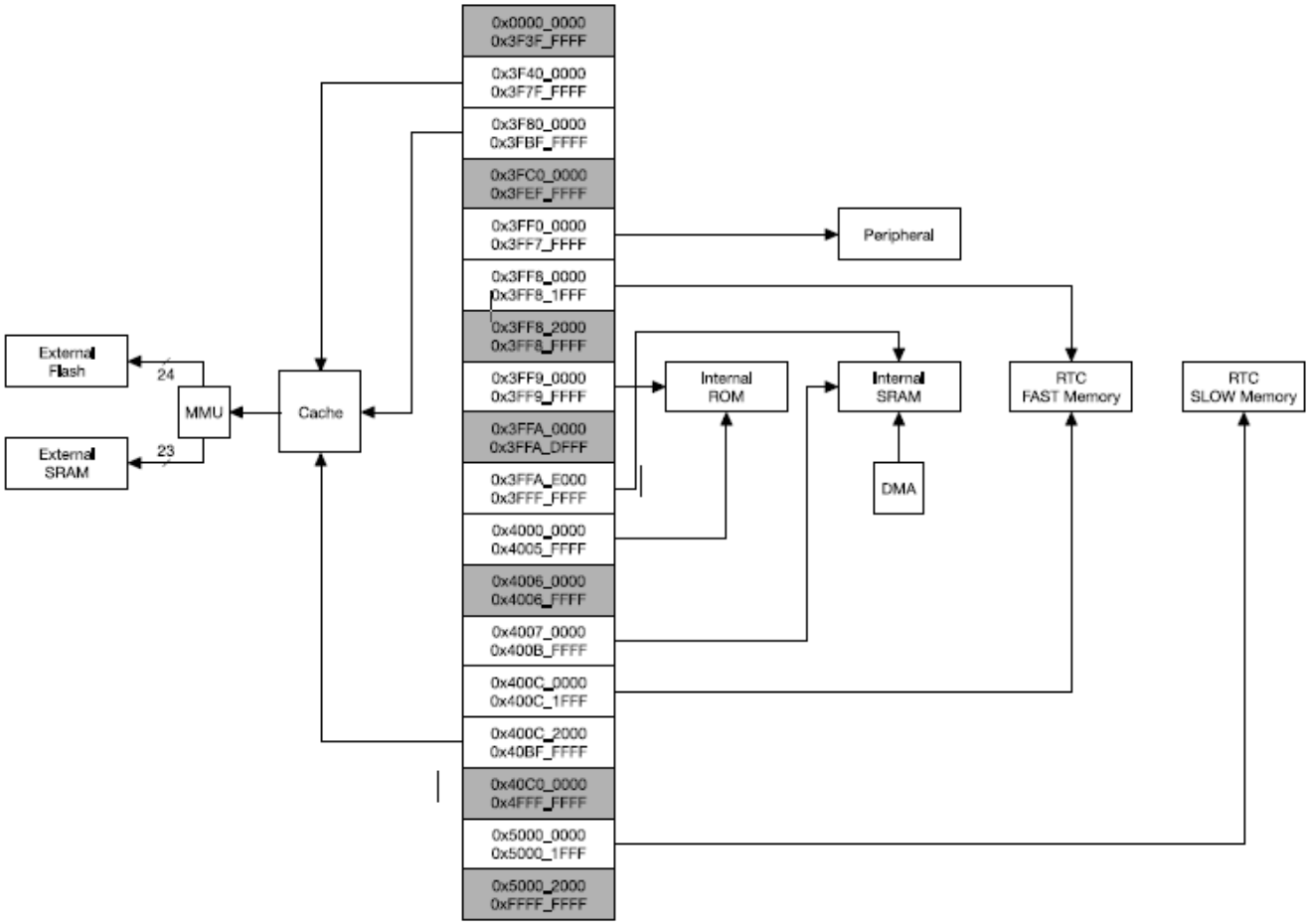
Reminder : MSP439F2274 Memory Map (64KB)

- 0x0000 to 0x01FF : peripherals
- 0x0200 to B=0x05FF : RAM (1KB), Data and Stack
- 0x0C00 to 0x0FFF : Boot mem (1KB, ROM).
- 0x1000 to 0x10FF : byte info. mem. (256 bytes, Flash)
- 0x8000 to 0xFFFF : Code (32 KB, Flash).
  - where : 0xFFE0 to 0xFFFF : interrupt vectors





ESP32 Memory Map (4GB address space)



- ESP32 memory map is slightly more complex...
- Peripherals adresses : from `0x3FF0_0000` to 0x3FF0_FFFF

ESP32 memory map excerpt

Category	Target	Start Address	End Address	Size
Embedded Memory	Internal ROM 0	0x4000_0000	0x4005_FFFF	384 KB
	Internal ROM 1	0x3FF9_0000	0x3FF9_FFFF	64 KB
	Internal SRAM 0	0x4007_0000	0x4009_FFFF	192 KB
	Internal SRAM 1	0x3FFE_0000	0x3FFF_FFFF	128 KB
		0x400A_0000	0x400B_FFFF	
	Internal SRAM 2	0x3FFA_E000	0x3FFD_FFFF	200 KB
	RTC FAST Memory	0x3FF8_0000	0x3FF8_1FFF	8 KB
		0x400C_0000	0x400C_1FFF	
	RTC SLOW Memory	0x5000_0000	0x5000_1FFF	8 KB
External Memory	External Flash	0x3F40_0000	0x3F7F_FFFF	4 MB
		0x400C_2000	0x40BF_FFFF	11 MB+248 KB
	External RAM	0x3F80_0000	0x3FBF_FFFF	4 MB
	DPort Register	0x3FF0_0000	0x3FF0_0FFF	4 KB
	AES Accelerator	0x3FF0_1000	0x3FF0_1FFF	4 KB
	RSA Accelerator	0x3FF0_2000	0x3FF0_2FFF	4 KB
	SHA Accelerator	0x3FF0_3000	0x3FF0_3FFF	4 KB
	Secure Boot	0x3FF0_4000	0x3FF0_4FFF	4 KB
	Cache MMU Table	0x3FF1_0000	0x3FF1_3FFF	16 KB
	PID Controller	0x3FF1_F000	0x3FF1_FFFF	4 KB
	UART0	0x3FF4_0000	0x3FF4_0FFF	4 KB
	SPI1	0x3FF4_2000	0x3FF4_2FFF	4 KB

- Excerpt of the ESP32 memory map (from `esp32_datasheet_en.pdf`)
- Configuration registers of peripheral “UART0” are accessed at addresses between `0x3FF4_0000` and `0x3FF4_0FFF`
- Our ESP32 model : 500KB embedded SRAM, 4 MB of external flash and 8MB of external PSRAM

Peripheral programming

- Peripherals are (nowadays) all programmed with *memory map*
  - Each peripheral contains configuration registers
  - These registers are *mapped* to special addresses in the memory
- Example : hardware multiplier of MSP430
  - Registers mapped between adresses `0x0130` et `0x013F`
  - Writing at adresse `0x130`, writes first operand
  - Writing at `0x138`, writes second operand and start the multiplication
  - The result is accessible by reading at address `0x013A` (on 32 bits)

MSP430 example of peripheral memory mapping

```
int main(void) {
    int i;
    int *p,*res;

    p=0x130;
    *p=2;
    p=0x138;
    *p=5;
    res=0x13A;
    i=*res;

    nop();
}
```

```
int main(void) {
    int i;
    int *p,*res;

    __asm__("mov #304, R4");
    __asm__("mov #2, @R4");
    // p=0x130;
    // *p=2;
    __asm__("mov #312, R4");
    __asm__("mov #5, @R4");
    // p=0x138;
    // *p=5;
    __asm__("mov #314, R4");
    __asm__("mov @R4, R5");
    // res=0x13A;
    i=*res;

    nop();
}
```

Use of Macros for Code Clarity

```
int main(void)   {
    int i;
    int *p,*res;

    p=0x130;
    *p=2;
    p=0x138;
    *p=5;
    res=0x13A;
    i=*res;

    nop();
}
```

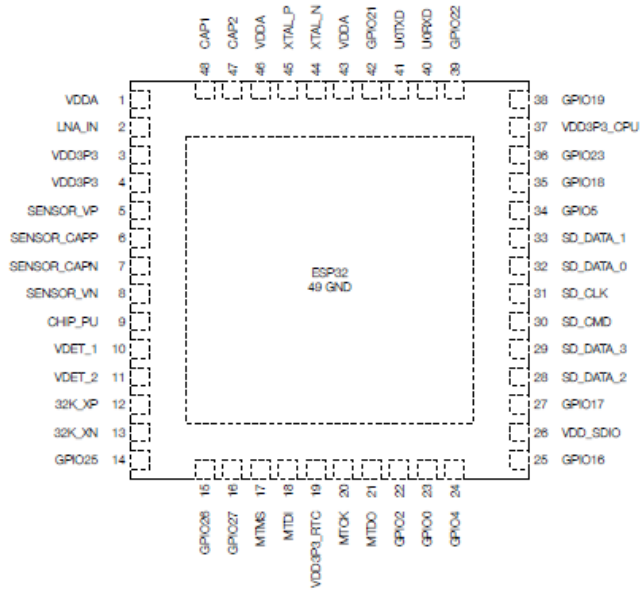
```
#include <themagicmacrofile.h>

int main(void) {
    int i;

    MULOP1=2;
    MULOP2=5;
    i=MULRES;

    nop();
}
```

Most basic peripheral : GPIO

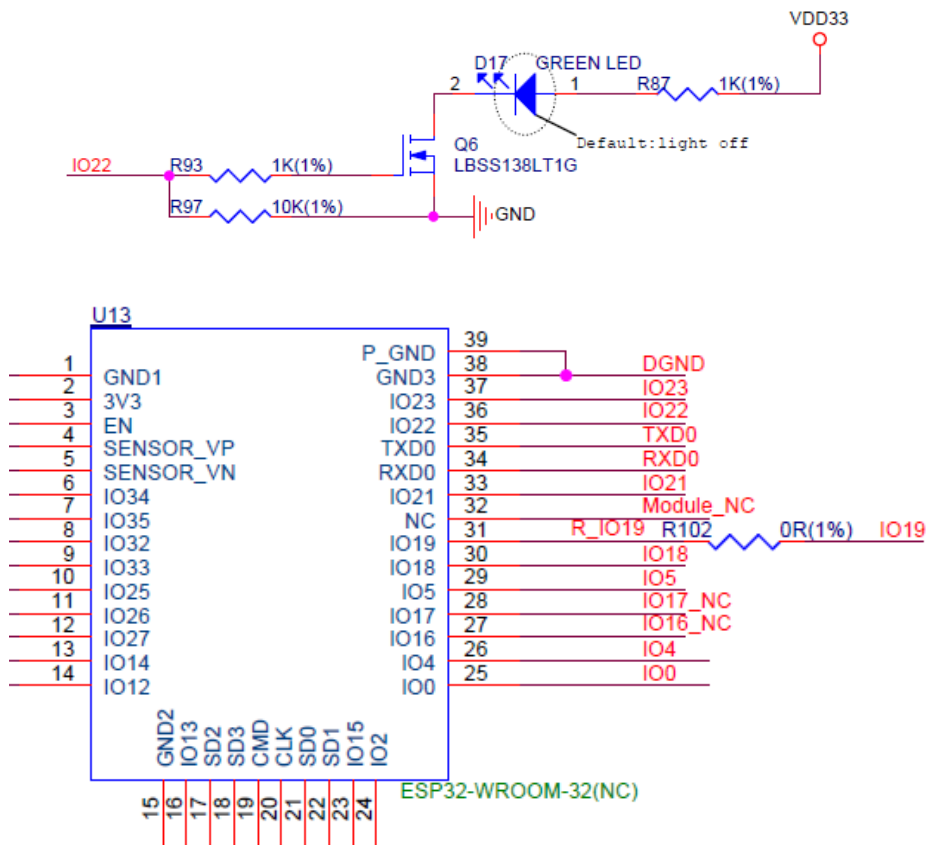


- ESP32 has 40 physical I/O pad
- None are accessible on LyraT except for the LED
- GPIO can be configured :
  - As input or output
  - Pulled up, pulled down, or not
  - Interrupt enable
  - The `IDF` example `examples/peripherals/gpio/` explains how to configure GPIOs

How to blink the LED on LyraT (1)

- Identify IO port connected to LED : LyraT schematics
  `https://dl.espressif.com/dl/schematics/esp32-lyrat-v4.3-schematic.pdf`

→ IO22



How to blink the LED on LyraT (2)

- Configure IO22 in output mode
- Write 1 or 0 at IO22 port address

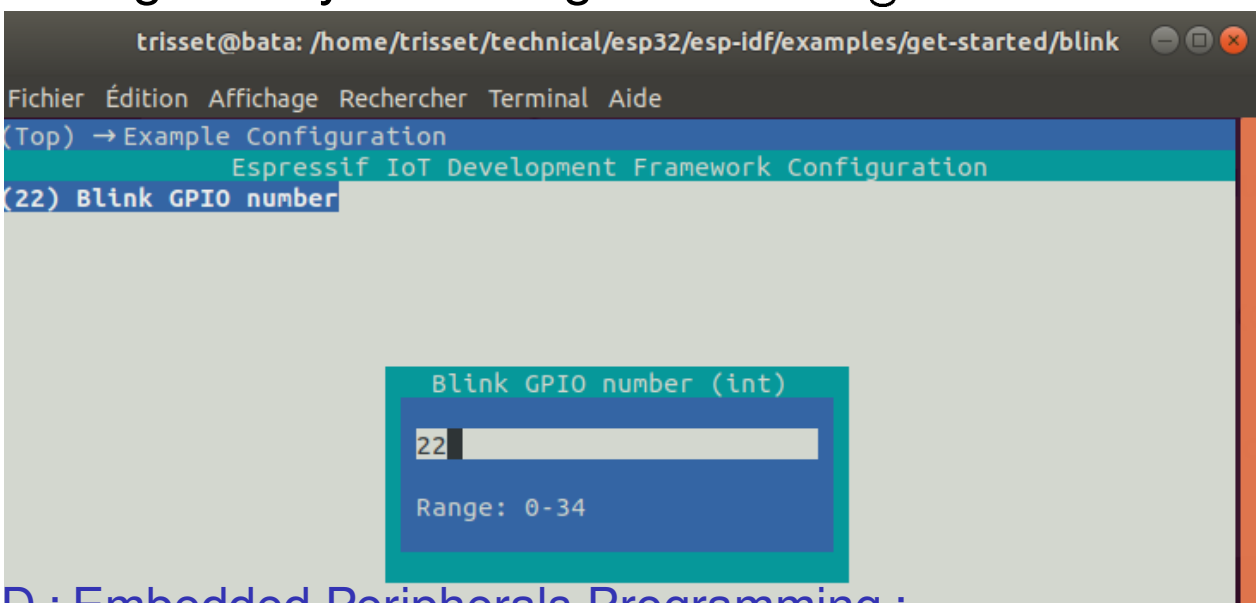
→ Use IDF API (`examples/get-started/blink/`)

```
gpio_reset_pin(BLINK_GPIO);
/* Set the GPIO as a push/pull output */
gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);
while(1) {
    /* Blink off (output low) */
    printf("Turning off the LED\n");
    gpio_set_level(BLINK_GPIO, 0);
    vTaskDelay(1000 / portTICK_PERIOD_MS);
    /* Blink on (output high) */
    printf("Turning on the LED\n");
    gpio_set_level(BLINK_GPIO, 1);
    vTaskDelay(1000 / portTICK_PERIOD_MS);
}
```

How to blink the LED on LyraT (3)

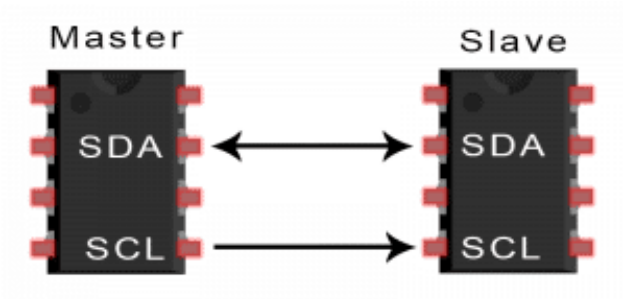
How and where are defined the macros `BLINK_GPIO` and `GPIO_MODE_OUTPUT` ?

- `GPIO_MODE_OUTPUT` is fixed independently of the board (value used to indicate that a GPIO is used as output) :
  - `components/soc/include/hal/gpio_types.h` :
    ```
    GPIO_MODE_OUTPUT_OD = ((GPIO_MODE_DEF_OUTPUT) | (GPIO_MO
    ```
  - `components/soc/soc/esp32/include/soc/gpio_caps.h` :
    ```
    #define GPIO_MODE_DEF_OUTPUT          (BIT1)
    ```
- `BLINK_GPIO` (i.e. port on which LED is connected) depends on the evaluation board
  - → Configured by `idf` using `menuconfig`



A more general peripheral : I2C

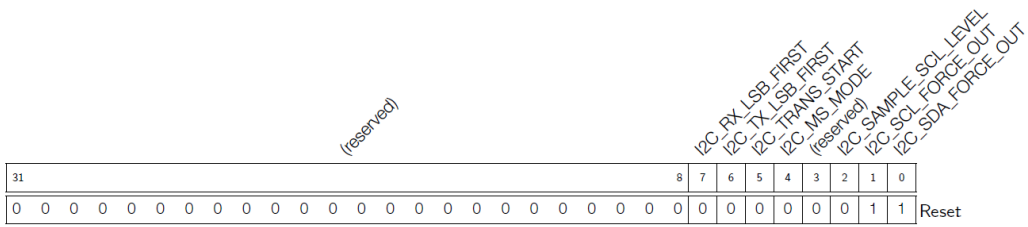
- I2C is a master/slave *synchronous* serial communication protocol
- It is used to communicate on both direction (R/W) bytes between master and slave
- *Synchronous* means that the clock synchronizing master and slave is sent by the master : no need of an agreement on transmission rate as in asynchronous protocol (such a UART : Universal Asynchronous Receiver Transmitter)
- I2C uses two wires : SCL (clock) and SDA (data)



How to understand how to use LyraT I2C

1. Learn I2C protocol (`https://fr.wikipedia.org/wiki/I2C`)
2. Read the ESP32 technical reference manual I2C chapter (`esp32_technical_reference_manual_en.pdf`, pp 284-307)
   - Each micro-controler has a dedicated hardware controler to handle I2C (i.e. the I2C *peripheral*)
   - This peripheral is control by a number of *registers*
   - The technical reference explains how to configure the registers in order to use the I2C peripheral, for instance :

Register 12.2: I2C_CTR_REG (0x0004)



I2C_RX_LSB_FIRST This bit is used to control the storage mode for received data. (R/W)
1: receive data from the least significant bit;
0: receive data from the most significant bit.

I2C_TX_LSB_FIRST This bit is used to control the sending mode for data needing to be sent. (R/W)
1: send data from the least significant bit;
0: send data from the most significant bit.

I2C_TRANS_START Set this bit to start sending the data in txfifo. (R/W)

I2C_MS_MODE Set this bit to configure the module as an I²C Master. Clear this bit to configure the module as an I²C Slave. (R/W)

I2C in brief (from SSM2603 codec doc)

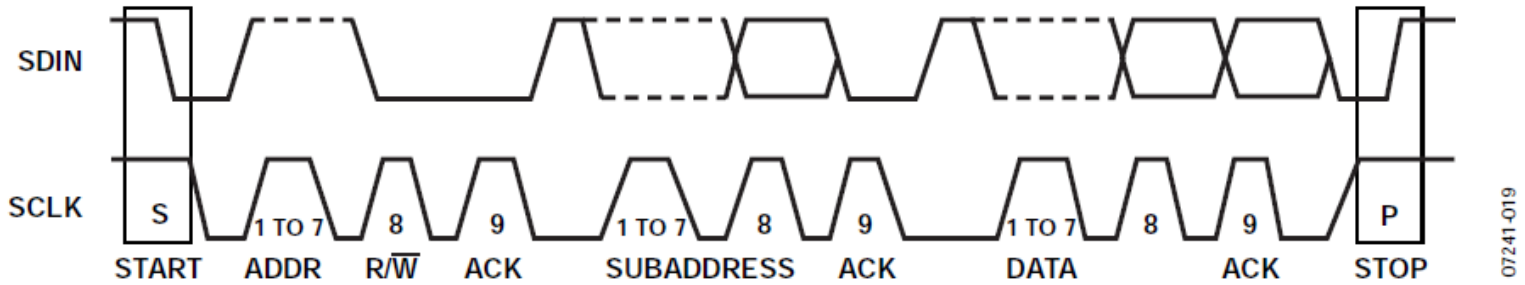
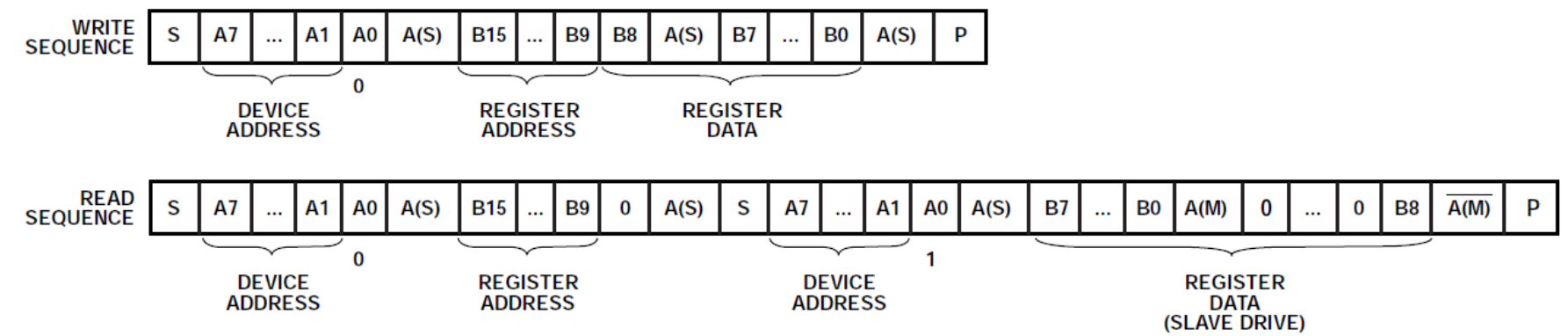


Figure 28. 2-Wire I²C Generalized Clocking Diagram



S/P = START/STOP BIT.
A0 = I²C R/$\overline{W}$ BIT.
A(S) = ACKNOWLEDGE BY SLAVE.
A(M) = ACKNOWLEDGE BY MASTER.
$\overline{A(M)}$ = ACKNOWLEDGE BY MASTER (INVERSION).

07241-022

Figure 29. I²C Write and Read Sequences

Use IDF I2C driver API (1)

examples/peripherals/i2c/i2c_self_test/i2c_example_main.c

```
static esp_err_t i2c_master_init(void)
{
    int i2c_master_port = I2C_MASTER_NUM;
    i2c_config_t conf;
    conf.mode = I2C_MODE_MASTER;
    conf.sda_io_num = I2C_MASTER_SDA_IO;
    conf.sda_pullup_en = GPIO_PULLUP_ENABLE;
    conf.scl_io_num = I2C_MASTER_SCL_IO;
    conf.scl_pullup_en = GPIO_PULLUP_ENABLE;
    conf.master.clk_speed = I2C_MASTER_FREQ_HZ;
    i2c_param_config(i2c_master_port, &conf);
    return i2c_driver_install(i2c_master_port, conf.mode, [..
}
```

Use IDF I2C driver API (2)

examples/peripherals/i2c/i2c_self_test/i2c_example_main.c

```
static esp_err_t i2c_master_write_slave(i2c_port_t i2c_num,
                                        uint8_t *data_wr, size_t size
{
    i2c_cmd_handle_t cmd = i2c_cmd_link_create();
    i2c_master_start(cmd);
    i2c_master_write_byte(cmd,
            (ESP_SLAVE_ADDR << 1) | WRITE_BIT, ACK_CHECK_EN);
    i2c_master_write(cmd, data_wr, size, ACK_CHECK_EN);
    i2c_master_stop(cmd);
    esp_err_t ret = i2c_master_cmd_begin(i2c_num, cmd,
                                         1000 / portTICK_RATE_MS);
    i2c_cmd_link_delete(cmd);
    return ret;
}
```

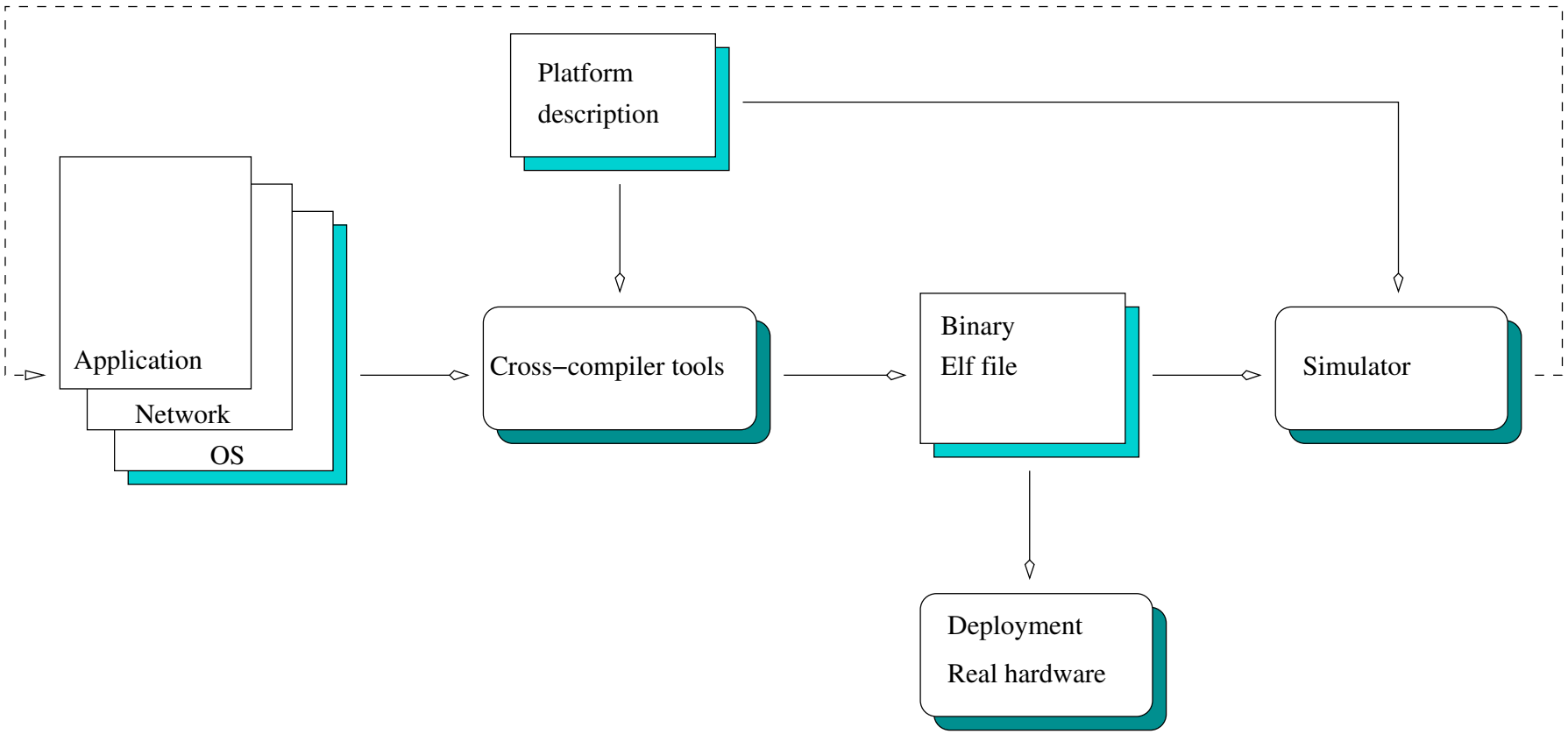
I2C and I2S on LyraT and 5TC-AUD

- On LyraT, I2C (and I2S) are used only to communicate with ES8388 audio codec.
- IDF propose a complex high level API that uses many other components of IDF and ADF.
  `esp-idf/components/driver/i2c.c`
  `esp-adf/components/audio_hal/driver/es8388`
- Romain choose to use a simpler driver (See 5TC-AUD lecture 4, `https://grame-cncm.github.io/embaudio20/lectures/lecture4/`) using a more classical peripheral configuration derived from the documentation of the ES8388 (`http://file1.dzsc.com/product/14/11/12/824138_145157417.pdf`
- Check file `ES8388` on AUD web site `https://github.com/grame-cncm/embaudio20/blob/master/examples/lib/ES8388.cpp`)

Compilation pour l'embarqué

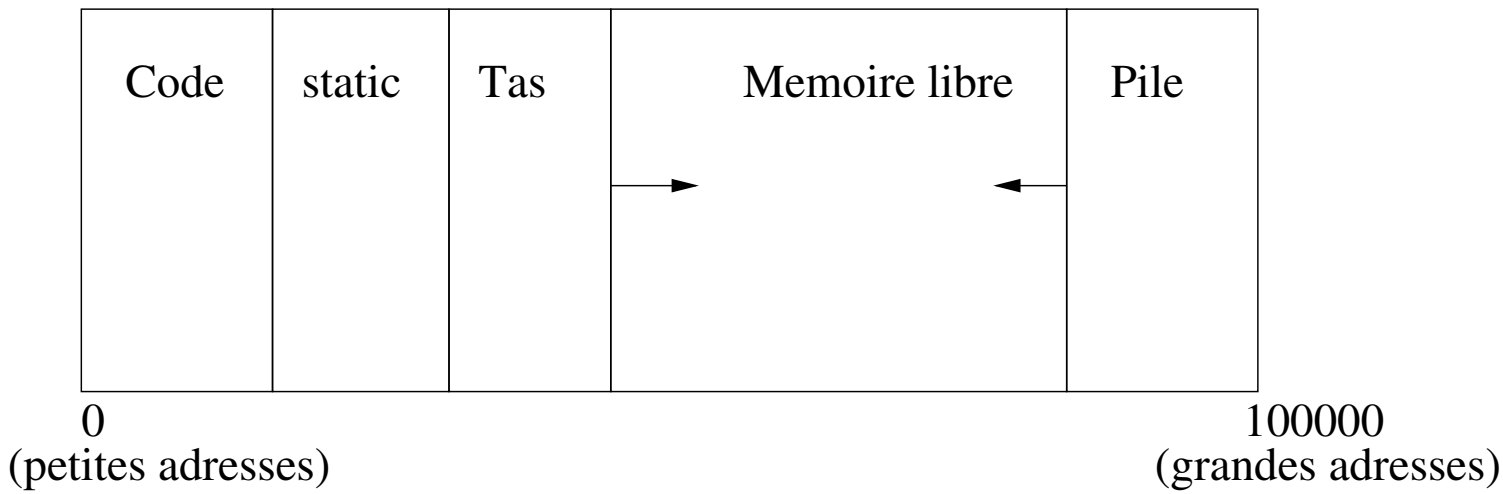
Rappels de compilation et fonctionnement des interruptions

Compilation



Pile d'exécution

- Le mécanisme de transfert de contrôle entre les procédures est implémenté grâce à la *pile d'exécution*.
- Le programmeur à cette vision de la mémoire virtuelle :

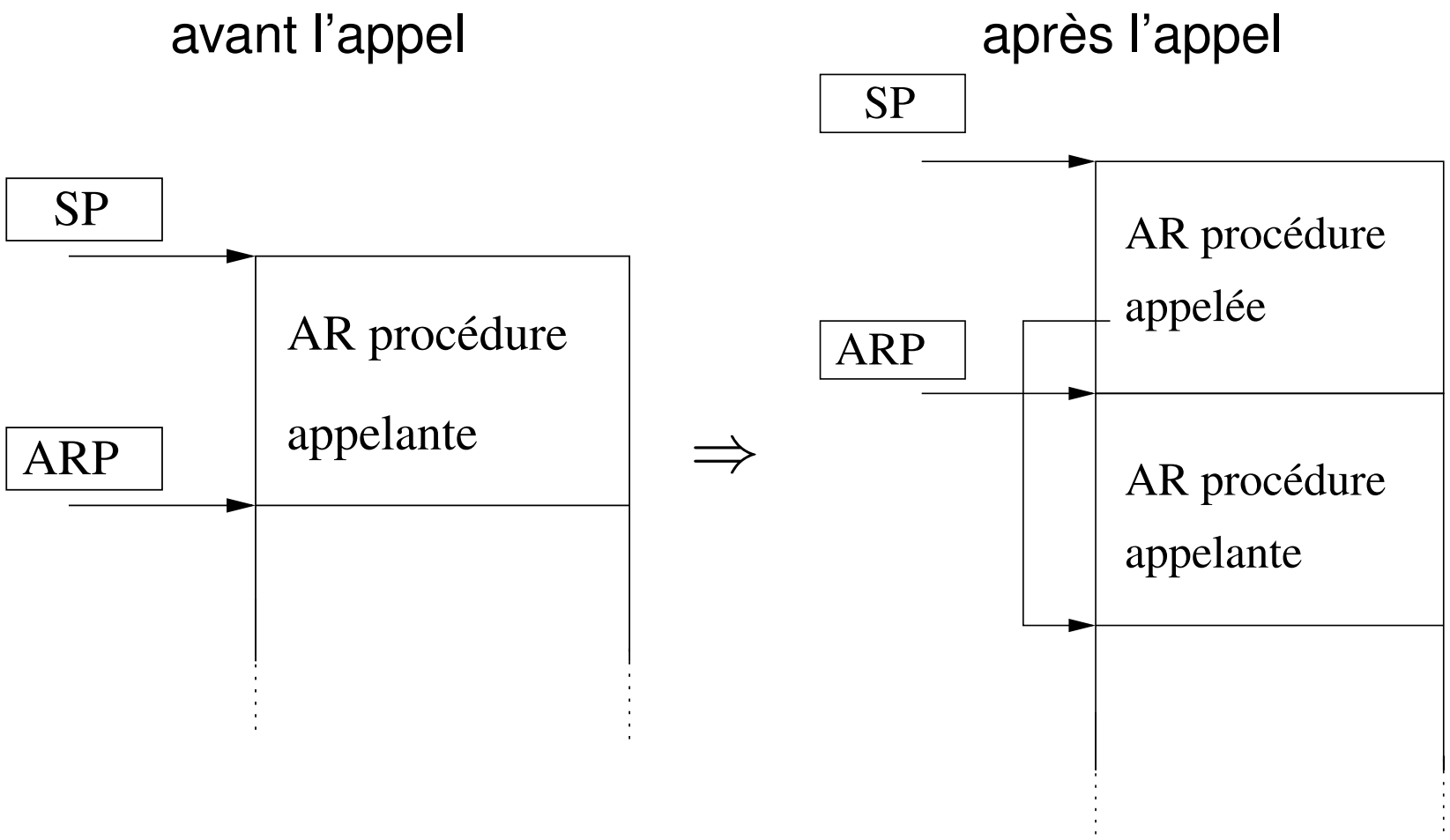


- Le tas (*heap*) est utilisé pour l'allocation dynamique.
- La pile (*stack*) est utilisée pour la gestion des contextes des procédures (variable locales, etc.)

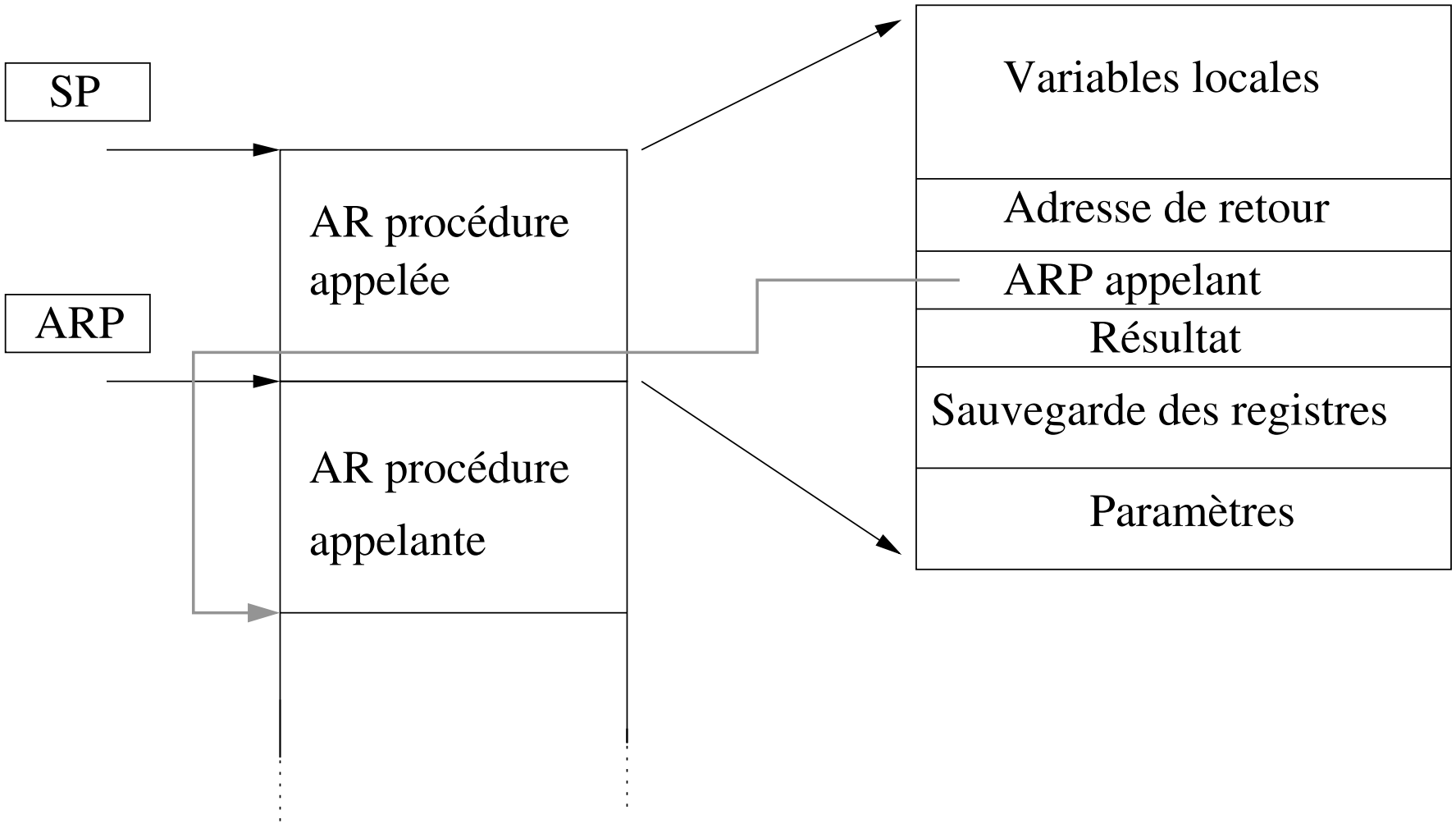
Enregistrement d'activation

- Appel d'une procédure : empilement de l'*enregistrement d'activation* (AR pour *activation record*).
- L'AR permet de mettre en place le *contexte* de la procédure.
- Cet AR contient
  - L'espace pour les variables locales déclarées dans la procédure
  - Des informations pour la restauration du contexte de la procédure appelante :
    - Pointeur sur l'AR de la procédure appelante (ARP ou FP pour *frame pointeur*).
    - Adresse de l'instruction de retour (instruction suivant l'appel de la procédure appelante).
    - Éventuellement sauvegarde de l'état des registres au moment de l'appel.

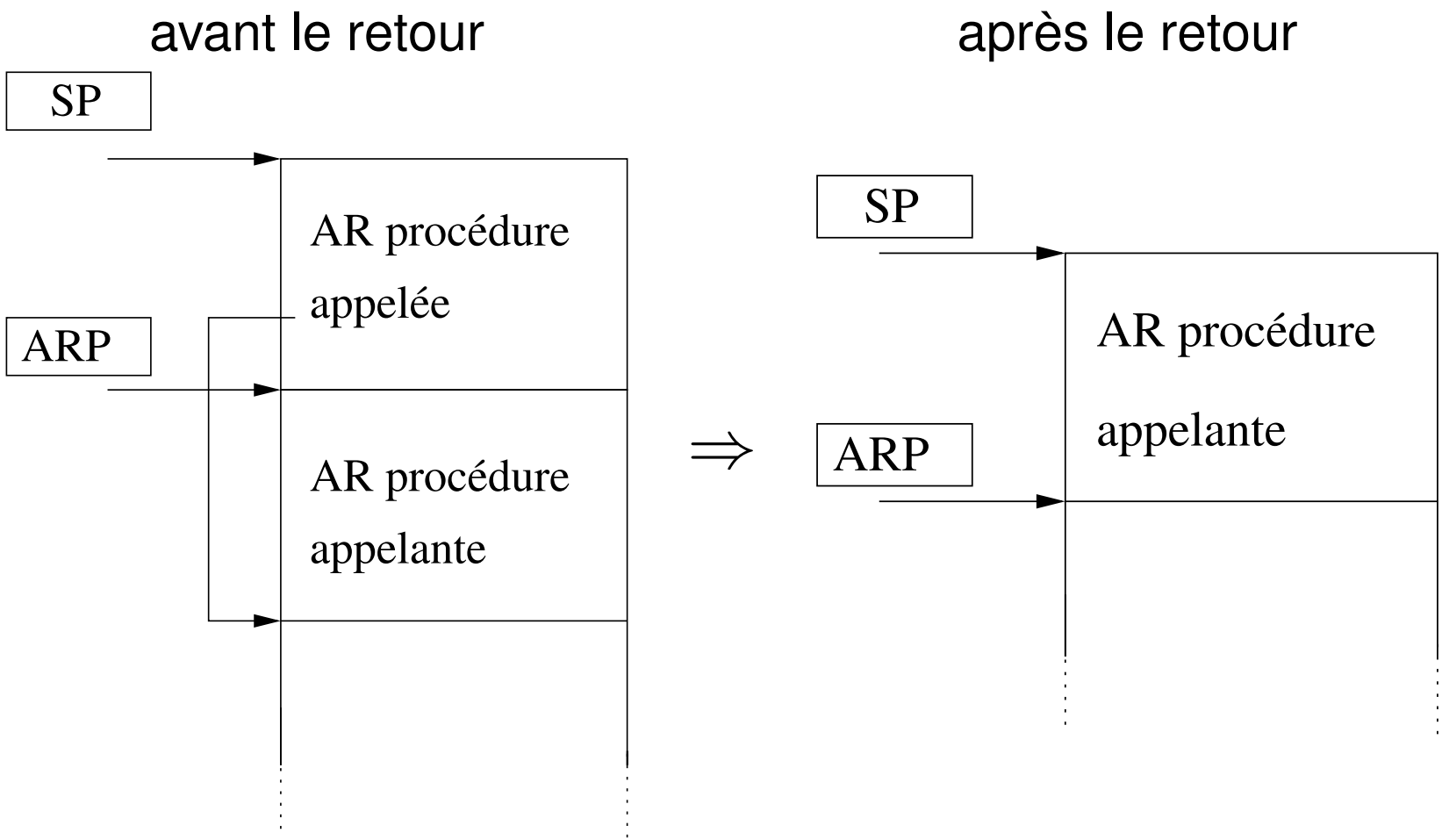
Appel de procédure : état de la pile



Contenu de l'AR

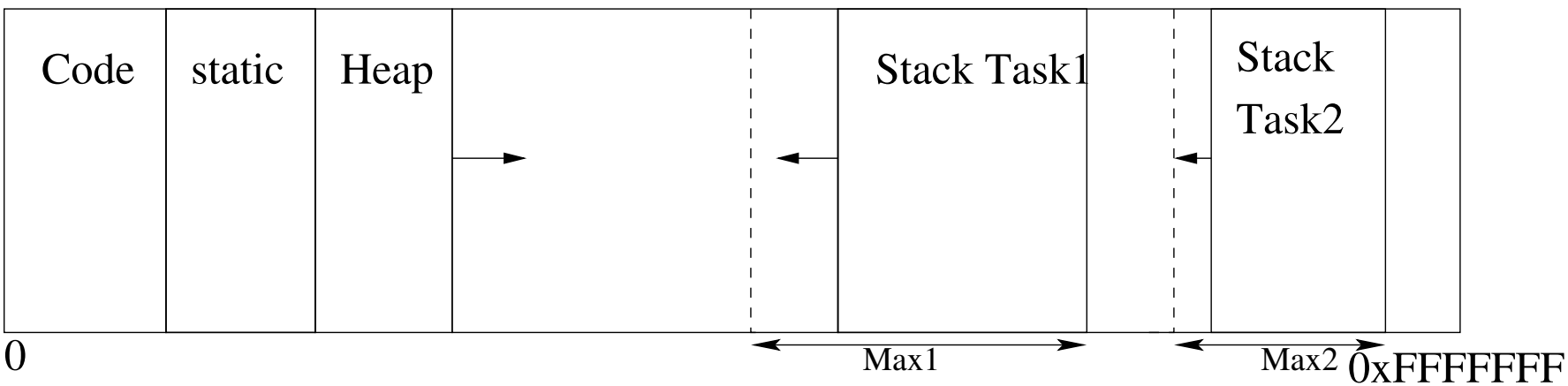


Retour de procédure : état de la pile



Qu'en est-t'il du multi-tâches ?

- Sur la pluspart des *gros* systèmes d'exploitation, il existe un mécanisme multi-tâches qui permet à plusieurs *processus* de s'exécuter de manière indépendante, i.e. chacun avec sa propre pile.
- En embarqué, c'est plus compliqué. on peut :
  - N'avoir qu'une tâche (*bare metal*) : la fonction `main`.
  - Avoir un petit système qui permet des pseudo tâches (co-routine) qui partagent la même pile
  - Avoir un petit système qui implémente un système de tâches rudimentaire avec piles séparées (exemple : freeRtos).



Principe du mécanisme d'interruption

- Par défaut, le programme `main` est exécuté à l'infini, il contient en général une boucle infinie pour ne jamais terminer.
- Le processeur peut recevoir à tout instant des *interruptions* (sous-entendu *interruptions matérielles*).
- Une interruption peut être envoyée par un périphérique du micro-controleur (timer, chip radio, port série, etc...), ou reçue de l'exterieur (sur un GPIO) comme le `reset` par exemple.
- C'est le programmeur qui configure les périphériques (par exemple le timer) pour envoyer un interruption lors de certains évennements
- Qu'elles soient internes ou externes les interruptions arrivent, par abus de language sur un *port* du micro-controleur.
- Elle est traitée par un *handler d'interruption* (ou *interrupt service routine* : ISR).
- Chaque interruption possède sa propre ISR. c'est une fonction écrite par le programmeur qui a des propriété un peu particulières.

Traitement d'une Interruption

- Les interruptions (sous-entendu "matérielles") sont indispensables au fonctionnement de tout ordinateur.

- Lorsqu'une interruption survient, le microprocesseur sauvegarde l'état courant de son programme en cours d'exécution :
  - tous les registre généraux
  - le registre d'état
  - le program counter

- Il exécute ensuite une portion de code spécifique pour traiter cette interruption (interrupt handler ou ISR)
- lorsque le handler est terminé, il restaure l'état du processeur et reprend l'exécution du programme interrompu

Interrupt Service Routine (ISR)

- L'appel à la routine de traitement de l'interruption n'est pas exactement un appel de fonction comme les autres.
- Elle doit être compilée de manière un peu différente, elle est donc en général identifiée par un *pragma* à destination du compilateur. Example pour `gcc` :
  `interrupt(PORT1_VECTOR)port1_irq_handler(void)`
- un handler d'interruption peut, lui même, être interrompu ou pas par une autre interruption (priorité des interruptions).
- L'utilisateur peut écrire ses propre routines d'interruption en C, les compilateurs fournissent des facilités pour cela.
- Sur les système un peu plus évolués l'ISR est fournit par l'environnement de programmation qui propose à l'utilisateur d'écrire une fonction qui sera appelées lors de l'interruption : mécanisme de *callback*

Callback mecanism

- A callback mecanism is used to allow the user to write its own ISR function
- In primitive systems (bare metal) :
  - The compiler uses pragmas to distinguish between regular function and ISR.
  - Each interrupt has a dedicated number corresponding to its entry in the *interrupt vector table*
- In more elaborate systems :
  - A function pointer mecanism is used to *register* a user fonction as callback for a given interrupt
  - Examples in IDF (`timer_group_example_main.c` and `touch_pad_interrupt` examples) :

```
timer_isr_register(TIMER_GROUP_0, timer_idx, timer_group0_isr
                   (void *) timer_idx, ESP_INTR_FLAG_IRAM, NUL

touch_pad_isr_register(tp_example_rtc_intr, NULL);
```