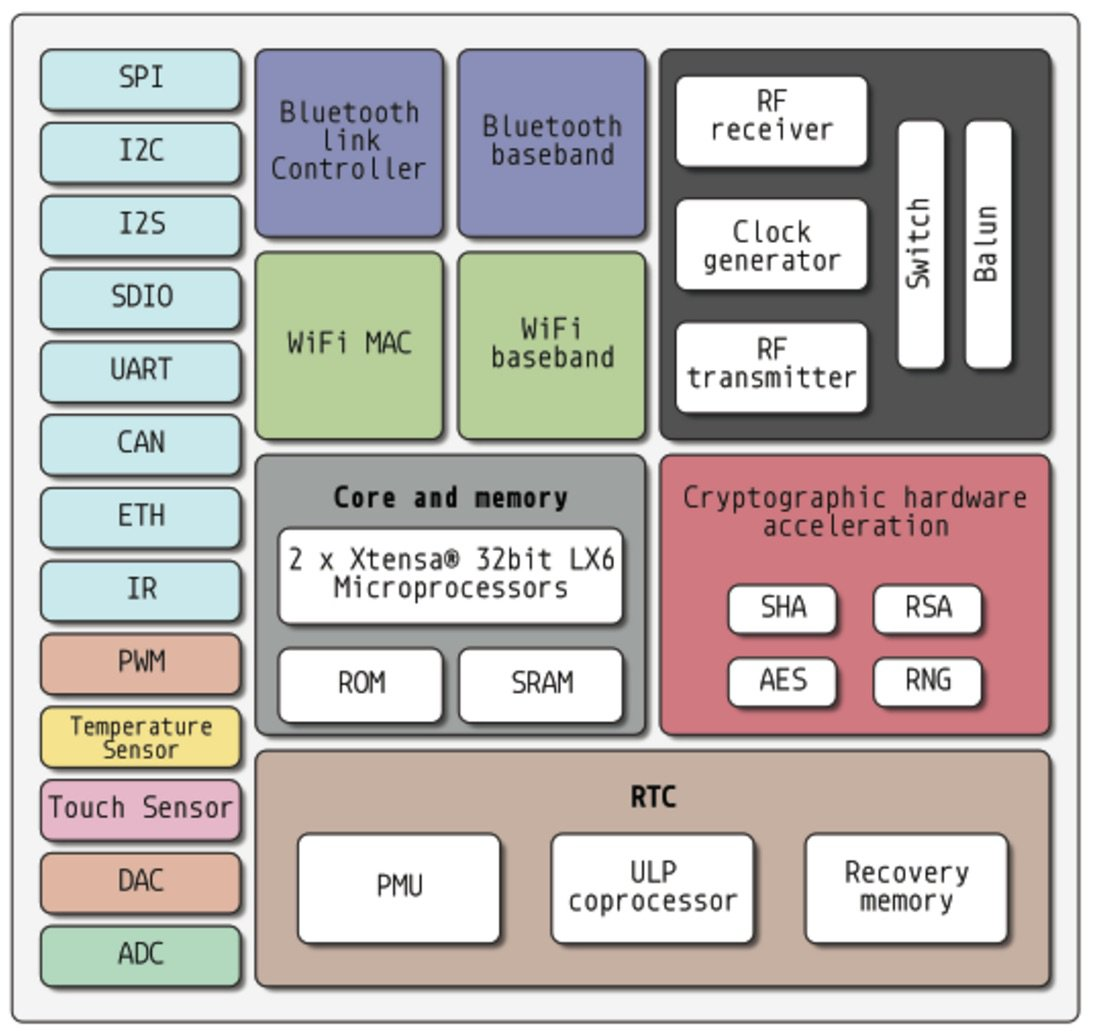
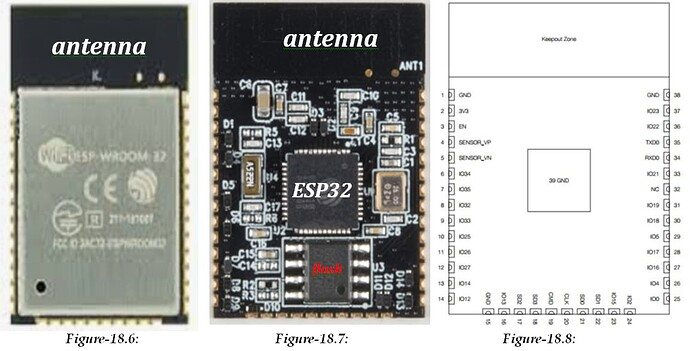
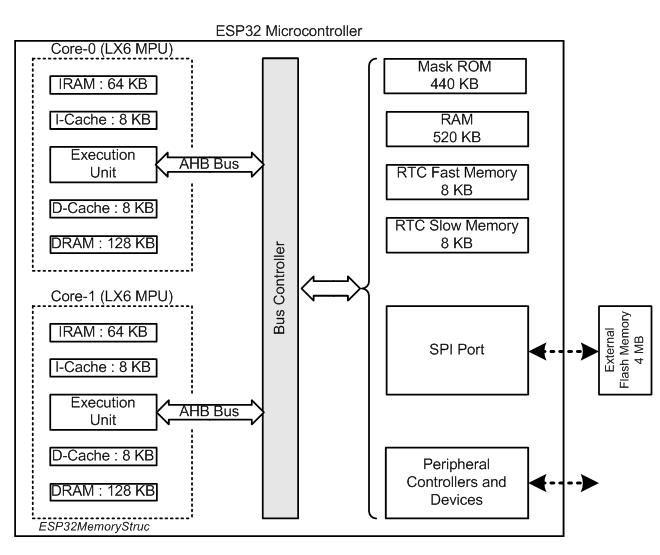
* 1. ESP32

The ESP32 (ESP-WROOM-32) module is based on the ESP32 SoC (System on Chip) by Espressif Systems. The ESP32 is powered by a dual-core 32-bit Xtensa LX6 microprocessor, which can run at a clock frequency of up to 240 MHz. It also includes a range of peripherals and features such as Wi-Fi, Bluetooth (both Classic and BLE), GPIO, ADCs, DACs, and various communication interfaces like SPI, I2C, UART, and more.







1-2- Programming Examples using ESP32 Dev Module

*#****include*** *"esp\_system.h"*

*#****include*** *"esp\_spi\_flash.h"*

void **setup**()

{

Serial.begin(9600);

**while**(!Serial)

{

;

}

Serial.println();

esp\_chip\_info\_t chip\_info;

esp\_chip\_info(&chip\_info); *//gives: chip\_info.model = 1, chip\_info.revision = 3*

uint32\_t flashSize = spi\_flash\_get\_chip\_size();

Serial.print("Off-chip Flash Memory Size: ");

Serial.print(flashSize/(1024\*1024)); Serial.println(" MB");

}

void **loop**() {}

1-3- Concurrent (independent) Blinking of Three LEDs - FreeRTOS-

*//function declarations*

TaskHandle\_t Task10Handle;

TaskHandle\_t Task11Handle;

*//GPIO definitions*

*#****define*** *LED 2*

*#****define*** *LED10 21*

*#****define*** *LED11 22*

*//task creation using FreeRTOS.h Librray functions*

void **setup**()

{

Serial.begin(9600);

pinMode(LED, **OUTPUT**);

pinMode(LED10, **OUTPUT**);

pinMode(LED11, **OUTPUT**);

xTaskCreatePinnedToCore(Task10, "Task-10", 2048, **NULL**, 1, &Task10Handle, 1);

xTaskCreatePinnedToCore(Task11, "Task-11", 2048, **NULL**, 1, &Task10Handle, 1);

}

*//LED blinks at 4000 ms interval*

void **loop**()

{

digitalWrite(LED, **HIGH**);

vTaskDelay(2000);

digitalWrite(LED, **LOW**);

vTaskDelay(2000);

}

*//LED10 blinks at 2000 ms interval*

void **Task10**(void \*pvParameters)

{

**while** (**true**)

{

digitalWrite(LED10, **HIGH**);

vTaskDelay(1000);

digitalWrite(LED10, **LOW**);

vTaskDelay(1000);

}

}

*//LED11 blinks at 1000 ms interval*

void **Task11**(void \*pvParameters)

{

**while** (**true**)

{

digitalWrite(LED11, **HIGH**);

vTaskDelay(500);

digitalWrite(LED11, **LOW**);

vTaskDelay(500);

}

}

2-1- Atmel ATSAM3X8E

The Atmel ATSAM3X8E is a microcontroller from Microchip Technology (previously Atmel), based on the ARM Cortex-M3 32-bit RISC processor. Here are some key features of the ATSAM3X8E:

-Processor Core: ARM Cortex-M3 running at up to 84 MHz.

-Flash Memory: 512 KB of in-system programmable Flash.

-SRAM: 96 KB (64 KB + 32 KB) of SRAM.

-Peripherals: Includes multiple communication interfaces like SPI, I2C, UART, CAN, and USB 2.0 OTG (On-The-Go).

-Analog Features: 12-bit ADC (Analog-to-Digital Converter) with up to 15 channels, and a 12-bit DAC (Digital-to-Analog Converter).

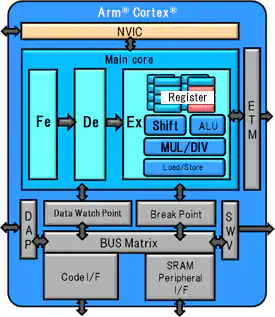
-Timers: Several timers including PWM (Pulse Width Modulation) and real-time clock (RTC).

-GPIO: Wide range of general-purpose I/O pins.

-Operating Voltage: 3.3V.

-Package: Available in various packages, including 100-pin TQFP (Thin Quad Flat Package).

The ATSAM3X8E is widely used in embedded systems, including applications like industrial automation, home automation, and in development boards like the Arduino Due, which is based on this microcontroller.



2-2- FreeRTOS for Arduino Due -ARM Cortex M3

**Ex.1**

This program is another FreeRTOS-based application designed for the Arduino Due. Its primary purpose is to demonstrate the basic creation and execution of tasks in a real-time operating system (RTOS).

#include "Arduino\_due\_FreeRTOS.h"

/\* Demo includes. \*/

#include "basic\_io\_arm.h"

/\* Used as a loop counter to create a very crude delay. \*/

#define mainDELAY\_LOOP\_COUNT ( 0xfffff )

/\* The task functions. \*/

void vTask1( void \*pvParameters );

void vTask2( void \*pvParameters );

/\*-----------------------------------------------------------\*/

void setup( void )

{

Serial.begin(9600);

/\* Create one of the two tasks. \*/

xTaskCreate( vTask1, /\* Pointer to the function that implements the task. \*/

"Task 1", /\* Text name for the task. This is to facilitate debugging only. \*/

200, /\* Stack depth - most small microcontrollers will use much less stack than this. \*/

NULL, /\* We are not using the task parameter. \*/

1, /\* This task will run at priority 1. \*/

NULL ); /\* We are not using the task handle. \*/

/\* Create the other task in exactly the same way. \*/

xTaskCreate( vTask2, "Task 2", 200, NULL, 1, NULL );

/\* Start the scheduler so our tasks start executing. \*/

vTaskStartScheduler();

/\* If all is well we will never reach here as the scheduler will now be

running. If we do reach here then it is likely that there was insufficient

heap available for the idle task to be created. \*/

for( ;; );

// return 0;

}

/\*-----------------------------------------------------------\*/

void vTask1( void \*pvParameters )

{

const char \*pcTaskName = "Task 1 is running\r\n";

volatile unsigned long ul;

/\* As per most tasks, this task is implemented in an infinite loop. \*/

for( ;; )

{

/\* Print out the name of this task. \*/

vPrintString( pcTaskName );

/\* Delay for a period. \*/

for( ul = 0; ul < mainDELAY\_LOOP\_COUNT; ul++ )

{

/\* This loop is just a very crude delay implementation. There is

nothing to do in here. Later exercises will replace this crude

loop with a proper delay/sleep function. \*/

}

}

}

/\*-----------------------------------------------------------\*/

void vTask2( void \*pvParameters )

{

const char \*pcTaskName = "Task 2 is running\r\n";

volatile unsigned long ul;

/\* As per most tasks, this task is implemented in an infinite loop. \*/

for( ;; )

{

/\* Print out the name of this task. \*/

vPrintString( pcTaskName );

/\* Delay for a period. \*/

for( ul = 0; ul < mainDELAY\_LOOP\_COUNT; ul++ )

{

/\* This loop is just a very crude delay implementation. There is

nothing to do in here. Later exercises will replace this crude

loop with a proper delay/sleep function. \*/

}

}

}

void loop() {}

**Ex.2**

The purpose of the program is to demonstrate task creation, inter-task communication, and the use of queues in a real-time operating system (RTOS) environment.

*Code by: FreeRTOS creator Richard Barry*

/\* FreeRTOS.org includes. \*/

#include "Arduino\_due\_FreeRTOS.h"

//#include "task.h"

//#include "semphr.h"

/\* Compiler includes. \*/

//#include <stdlib.h>

//#include <stdio.h>

/\* The task that sends messages to the stdio gatekeeper. Two instances of this

task are created. \*/

static void prvPrintTask( void \*pvParameters );

/\* The gatekeeper task itself. \*/

static void prvStdioGatekeeperTask( void \*pvParameters );

/\* Define the strings that the tasks and interrupt will print out via the gatekeeper. \*/

static char \*pcStringsToPrint[] =

{

"Task 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\r\n",

"Task 2 ----------------------------------------------------\r\n",

"Message printed from the tick hook interrupt ##############\r\n"

};

/\*-----------------------------------------------------------\*/

/\* Declare a variable of type QueueHandle\_t. This is used to send messages from

the print tasks to the gatekeeper task. \*/

QueueHandle\_t xPrintQueue;

void setup( void )

{

Serial.begin(9600);

/\* Before a queue is used it must be explicitly created. The queue is created

to hold a maximum of 5 character pointers. \*/

xPrintQueue = xQueueCreate( 5, sizeof( char \* ) );

/\* The tasks are going to use a pseudo random delay, seed the random number

generator. \*/

srand( 567 );

/\* Check the queue was created successfully. \*/

if( xPrintQueue != NULL )

{

/\* Create two instances of the tasks that send messages to the gatekeeper.

The index to the string they attempt to write is passed in as the task

parameter (4th parameter to xTaskCreate()). The tasks are created at

different priorities so some pre-emption will occur. \*/

xTaskCreate( prvPrintTask, "Print1", 200, ( void \* ) 0, 1, NULL );

xTaskCreate( prvPrintTask, "Print2", 200, ( void \* ) 1, 2, NULL );

/\* Create the gatekeeper task. This is the only task that is permitted

to access standard out. \*/

xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 200, NULL, 0, NULL );

/\* Start the scheduler so the created tasks start executing. \*/

vTaskStartScheduler();

}

/\* If all is well we will never reach here as the scheduler will now be

running the tasks. If we do reach here then it is likely that there was

insufficient heap memory available for a resource to be created. \*/

for( ;; );

// return 0;

}

/\*-----------------------------------------------------------\*/

static void prvStdioGatekeeperTask( void \*pvParameters )

{

char \*pcMessageToPrint;

/\* This is the only task that is allowed to write to the terminal output.

Any other task wanting to write to the output does not access the terminal

directly, but instead sends the output to this task. As only one task

writes to standard out there are no mutual exclusion or serialization issues

to consider within this task itself. \*/

for( ;; )

{

/\* Wait for a message to arrive. \*/

xQueueReceive( xPrintQueue, &pcMessageToPrint, portMAX\_DELAY );

/\* There is no need to check the return value as the task will block

indefinitely and only run again when a message has arrived. When the

next line is executed there will be a message to be output. \*/

//printf( "%s", pcMessageToPrint );

//fflush( stdout );

Serial.print(pcMessageToPrint );

Serial.flush();

if (Serial.available()) {

vTaskEndScheduler();

}

/\* Now simply go back to wait for the next message. \*/

}

}

/\*-----------------------------------------------------------\*/

extern "C"{ // FreeRTOS expects C linkage

void vApplicationTickHook( void )

{

static int iCount = 0;

portBASE\_TYPE xHigherPriorityTaskWoken = pdFALSE;

/\* Print out a message every 200 ticks. The message is not written out

directly, but sent to the gatekeeper task. \*/

iCount++;

if( iCount >= 200 )

{

/\* In this case the last parameter (xHigherPriorityTaskWoken) is not

actually used but must still be supplied. \*/

xQueueSendToFrontFromISR( xPrintQueue, &( pcStringsToPrint[ 2 ] ), &xHigherPriorityTaskWoken );

/\* Reset the count ready to print out the string again in 200 ticks

time. \*/

iCount = 0;

}

}

}

/\*-----------------------------------------------------------\*/

static void prvPrintTask( void \*pvParameters )

{

int iIndexToString;

/\* Two instances of this task are created so the index to the string the task

will send to the gatekeeper task is passed in the task parameter. Cast this

to the required type. \*/

iIndexToString = ( int ) pvParameters;

for( ;; )

{

/\* Print out the string, not directly but by passing the string to the

gatekeeper task on the queue. The queue is created before the scheduler is

started so will already exist by the time this task executes. A block time

is not specified as there should always be space in the queue. \*/

xQueueSendToBack( xPrintQueue, &( pcStringsToPrint[ iIndexToString ] ), 0 );

/\* Wait a pseudo random time. Note that rand() is not necessarily

re-entrant, but in this case it does not really matter as the code does

not care what value is returned. In a more secure application a version

of rand() that is known to be re-entrant should be used - or calls to

rand() should be protected using a critical section. \*/

vTaskDelay( ( rand() & 0x1FF ) );

}

}

//------------------------------------------------------------------------------

void loop() {}

3-1- ATMEGA2560

The ATmega2560 is an 8-bit microcontroller from Microchip Technology, originally developed by Atmel. It is part of the AVR family of microcontrollers and is well-known for its use in the Arduino Mega 2560 development board. Here are some key features of the ATmega2560:

**Core:**

8-bit AVR RISC architecture: The ATmega2560 operates on an 8-bit Reduced Instruction Set Computing (RISC) architecture, which allows for efficient execution of instructions.

**Clock Speed:**

16 MHz: The maximum operating frequency is 16 MHz when using an external crystal oscillator.

**Memory:**

Flash Memory: 256 KB of in-system self-programmable Flash memory for storing programs.

SRAM: 8 KB of static RAM for data storage during program execution.

EEPROM: 4 KB of EEPROM for non-volatile data storage, which retains information even when the power is turned off.

**I/O Ports:**

86 General Purpose I/O Lines: The microcontroller has a large number of GPIO pins, making it suitable for projects that require a lot of input/output operations.

**Timers/Counters:**

6 timers: Three 16-bit timers and three 8-bit timers for generating accurate time delays or frequencies, counting events, or PWM generation.

**Communication Interfaces:**

-USART: Four USART (Universal Synchronous and Asynchronous serial Receiver and Transmitter) interfaces for serial communication.

-SPI: Serial Peripheral Interface for fast, synchronous data transfer between microcontrollers and other peripherals.

-I2C (TWI): Two-wire Interface for communication with other I2C-compatible devices.

-CAN: Controller Area Network interface (though this is less commonly used in hobbyist applications).

**Analog Features:**

16-channel 10-bit ADC (Analog-to-Digital Converter): Converts analog signals to digital values.

Analog Comparator: Compares two analog voltages and provides a digital output based on the comparison.

**PWM (Pulse Width Modulation):**

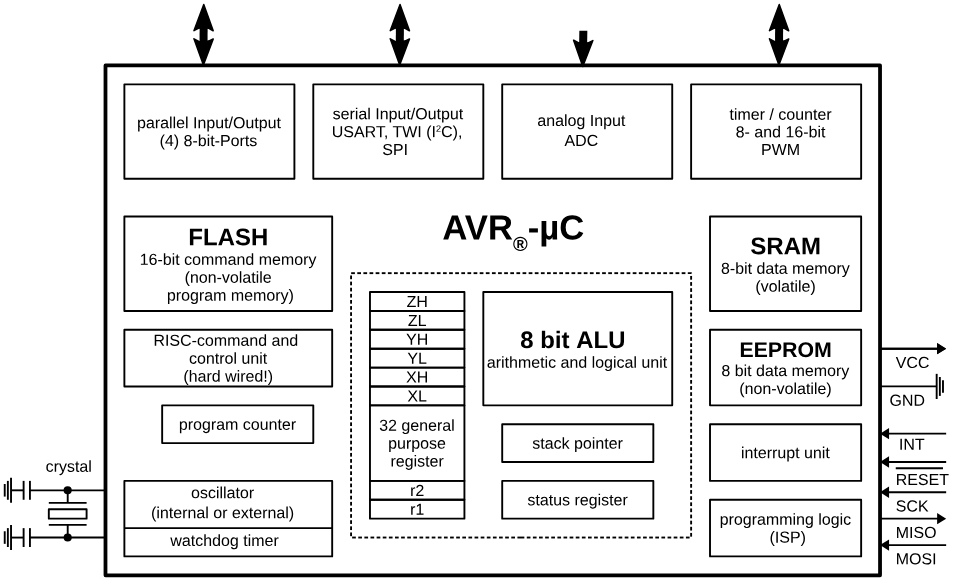
Up to 15 PWM channels, useful for controlling motors, LEDs, and other devices that require varying levels of power.

**Interrupts:**

External and internal interrupts: Multiple interrupt sources for handling external signals or internal events, which allows the microcontroller to respond to events in real-time.

**Power Consumption:**

The ATmega2560 is designed for low-power consumption, with different power-saving modes available to extend battery life in portable applications.



3-2- FreeRTOS for Arduino MEGA -ATMEGA 2560

**1.EX**

/\*

Example of a FreeRTOS mutex

https://www.freertos.org/Real-time-embedded-RTOS-mutexes.html

\*/

// Include Arduino FreeRTOS library

#include <Arduino\_FreeRTOS.h>

// Include mutex support

#include <semphr.h>

/\*

Declaring a global variable of type SemaphoreHandle\_t

\*/

SemaphoreHandle\_t mutex;

int globalCount = 0;

void setup() {

Serial.begin(9600);

/\*\*

Create a mutex.

https://www.freertos.org/CreateMutex.html

\*/

mutex = xSemaphoreCreateMutex();

if (mutex != NULL) {

Serial.println("Mutex created");

}

/\*\*

Create tasks

\*/

xTaskCreate(TaskMutex, // Task function

"Task1", // Task name for humans

128,

1000, // Task parameter

1, // Task priority

NULL);

xTaskCreate(TaskMutex, "Task2", 128, 1000, 1, NULL);

}

void loop() {}

void TaskMutex(void \*pvParameters)

{

TickType\_t delayTime = \*((TickType\_t\*)pvParameters); // Use task parameters to define delay

for (;;)

{

/\*\*

Take mutex

https://www.freertos.org/a00122.html

\*/

if (xSemaphoreTake(mutex, 10) == pdTRUE)

{

Serial.print(pcTaskGetName(NULL)); // Get task name

Serial.print(", Count read value: ");

Serial.print(globalCount);

globalCount++;

Serial.print(", Updated value: ");

Serial.print(globalCount);

Serial.println();

/\*\*

Give mutex

https://www.freertos.org/a00123.html

\*/

xSemaphoreGive(mutex);

}

vTaskDelay(delayTime / portTICK\_PERIOD\_MS);

}

}