Input / Output

Date	@November 14, 2024

An embedded system must interact with the external world; it does it using inputs and outputs.

Inputs:

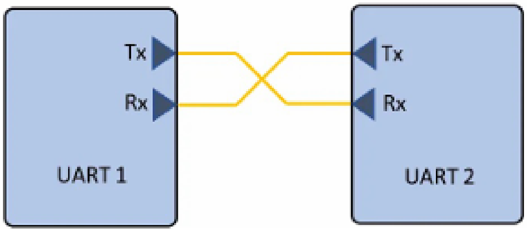
- sense the external world, measuring physical quantities
- react to user input
- receive commands or communications

Outputs:

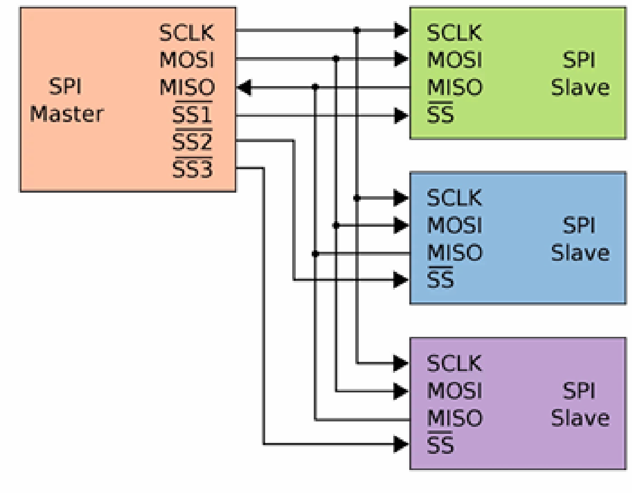
- drive external actuators
- signal events
- transmit information

Typical I/O in an embedded system

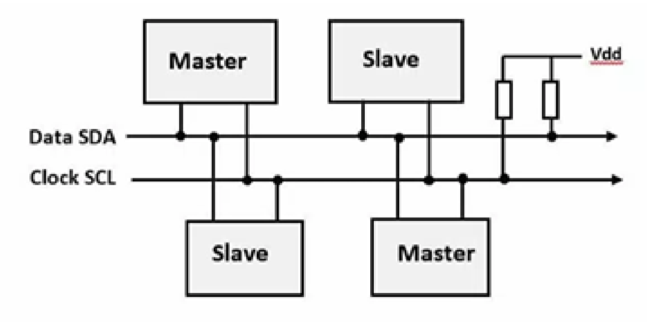
UART stands for universal asynchronous receiver transmitter and it uses a serial and asynchronous protocol. The number of wires used is 2, one for transmission and one for reception, and its speed is low-medium but configurable. It is typically used for point-to-point communication for debugging, data transfer between microcontroller and PC or communication with low-speed modules.



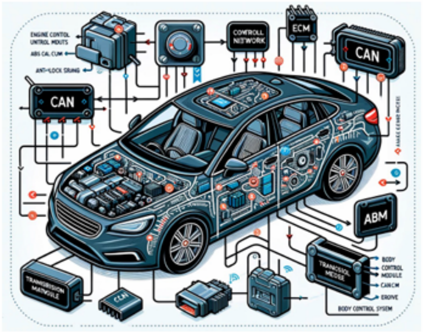
SPI stands for serial peripheral interface and it is a serial synchronous protocol. It uses 4 wires, one for te Master-In/Slave-Out, one for the Master-Out/Slave-In, one for the slave select and one for the clock. This protocol is very fast, in fact it is used for high-speed communication with flash memory, with sensors, analog to digital and digital to analog converters, and with displays.



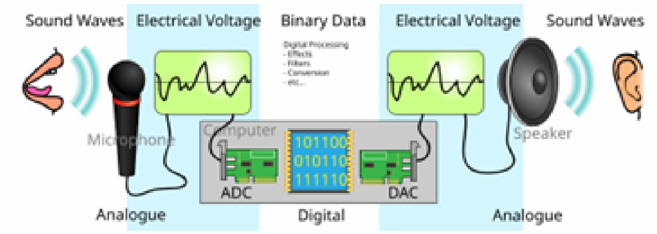
I2C stands for inter integrated circuits and is a serial, synchronous, multi-master and multi-slave protocol. It uses 2 wires, one for serial data and one for serial clock. It is used for low-speed communication between multiple devices or for connection of sensors, displays and flash memory to microcontrollers.



CAN stands for controller area network and is a serial, synchronous and multi-master protocol. It uses 2 wires, one for CAN high and one for CAN low. It is typically used for automotive and industrial systems, communication between electronic control units and when a reliable, noise resistant data transfer is needed.



ADC/DCA that is analog to digital / digital to analog converter are protocols that sample analog signals to convert them in the digital domain and viceversa. Their speed is configurable and depends on the input signal.



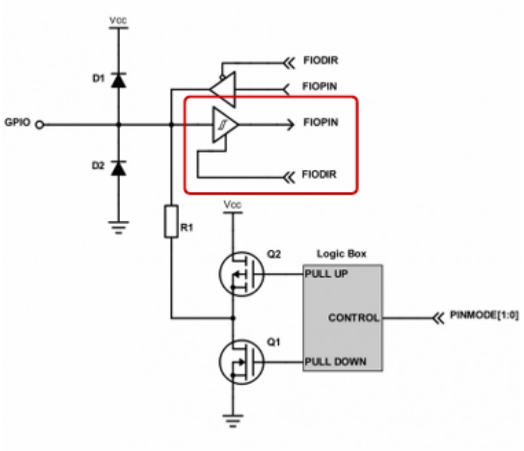
GPIO stands for general purpose input-output and it is a digital I/O controllable by software which can be configured to generate interrupts. It is configurable as input or output as well. It is used for controlling LEDs, reading button states, sending control signals, etc.

GPIO

A GPIO is versatile and configurable I/O interface, it is an essential building block to interface with external components.

INPUTE MODE

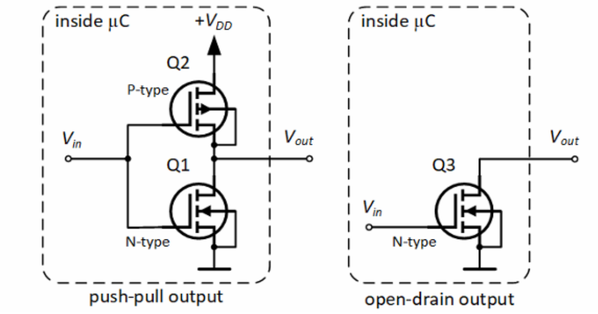
In input mode the GPIO is configured to read a signal from an external source and generally works with digital input. In some microntrollers can be connected to an internal ADC and used as an analog input. It is an high-impedance input, which means it does not affect the external driving circuit.



It includes the concept of hysteresis, which means that there is a slight difference between voltage levels at which the pin switches from high to low and from low to high. It reduces noise sensitivity and avoids fast switching when signal is around the threshold. Hysteresis is used when connecting mechanical switches which can have bounces, or sensors with noisy output.

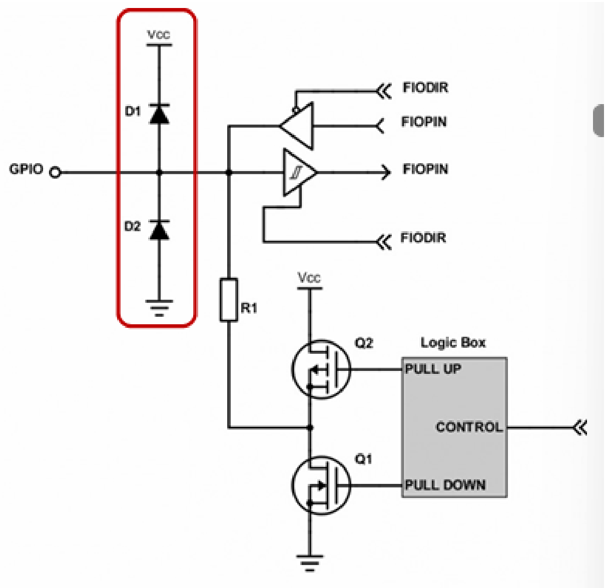
OUTPUT MODE

In output mode it drives an output signal. In the PUSH-PULL configuration it can drive both high and low states, while in OPEN-DRAIN configuration it can only pull the signal low. The latter is useful to create shared lines with multiple devices and to avoid conflicts, and requires external pull-up resistor.



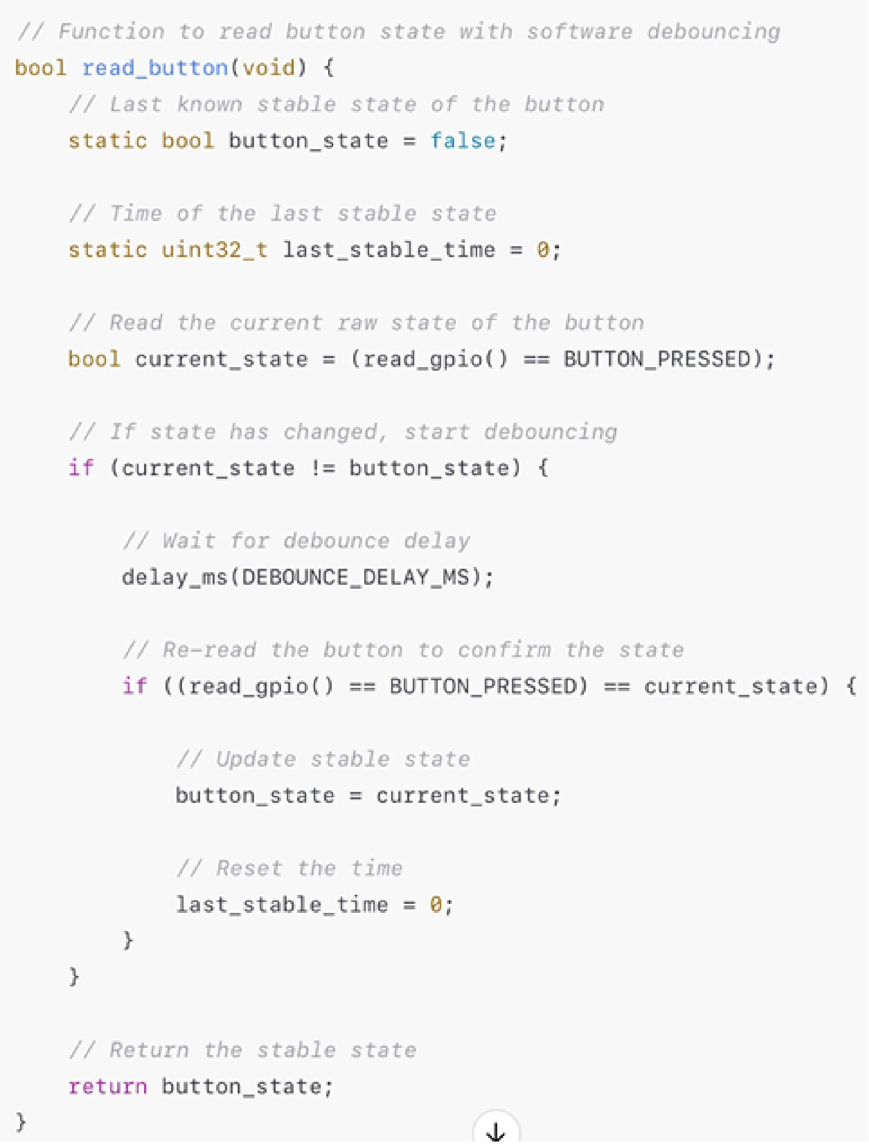
INPUT PROTECTION

For input protection clamping diodes can be used, and they shunt over-voltages to protect the rest of the circuit. D1 and D2 only turn on if GPIO input go over Vcc or below GND. GPIO is generally designed to work with small currents. An external protection, like resistors to limit the current, is generally required.



HARDWARE DEBOUNCING

Some boards allows hardware debouncing, which is a mechanical switch that, when pressed, the physical contact can bounce, causing fluctuation in the input signal. This can use unwanted reads on the GPIO pins. If hardware debouncing is not supported, a software debouncing must be implemented.



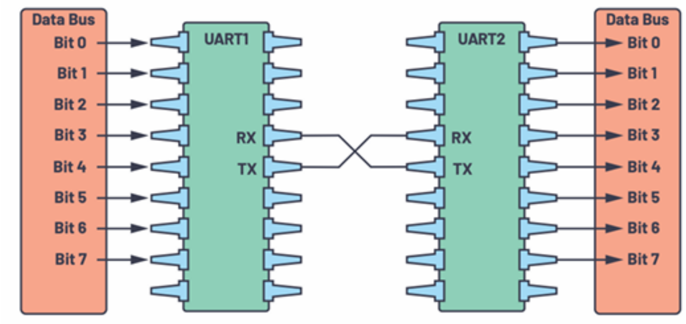
```c
// Function to read button state with software debouncing
bool read_button(void) {
    // Last known stable state of the button
    static bool button_state = false;

    // Time of the last stable state
    static uint32_t last_stable_time = 0;

    // Read the current raw state of the button
    bool current_state = (read_gpio() == BUTTON_PRESSED);

    // If state has changed, start debouncing
    if (current_state != button_state) {

        // Wait for debounce delay
        delay_ms(DEBOUNCE_DELAY_MS);

        // Re-read the button to confirm the state
        if ((read_gpio() == BUTTON_PRESSED) == current_state) {

            // Update stable state
            button_state = current_state;

            // Reset the time
            last_stable_time = 0;
        }
    }

    // Return the stable state
    return button_state;
}
```

Other features of GPIO are:

- alternate functions which use the GPIO to perform something different from simple I/O
- GPIO multiplexing, which allows to serve multiple alternate functions from the same GPIO
- interrupts, because the GPIO can be associated with an interrupt which is triggered when an event is measured on the GPIO

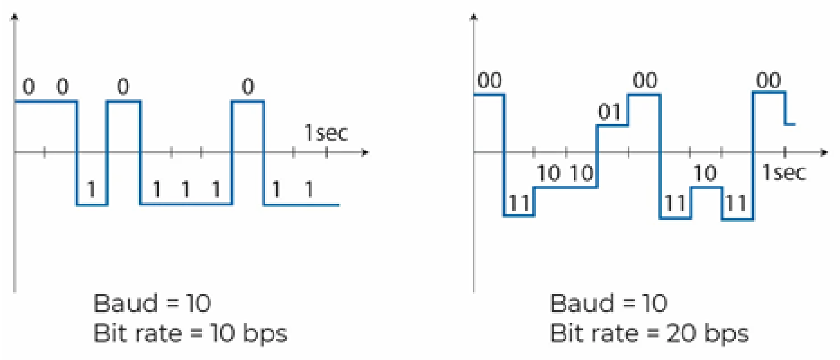
UART

UART is internally connected to a bus which sends data in parallel, serializing it.



BAUD RATE CONFIGURATION

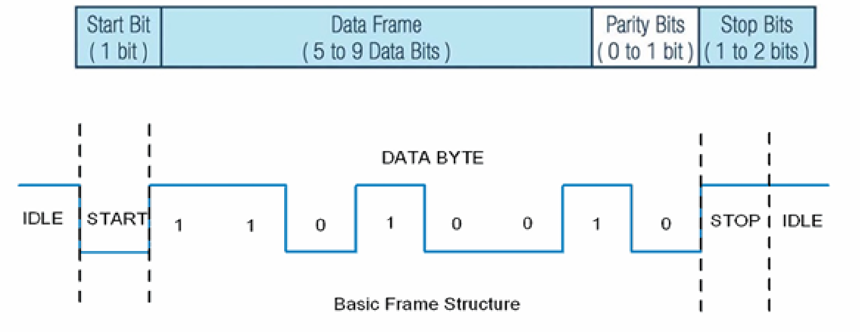
Baud rate is the more general term used to identify the number of symbols that can be sent each second. In a serial protocol the baud rate and bit rate are the same.



The transmitter and the receiver must use the same baud rate

DATA TRANSMISSION

In UART, transmission is done in form of packets

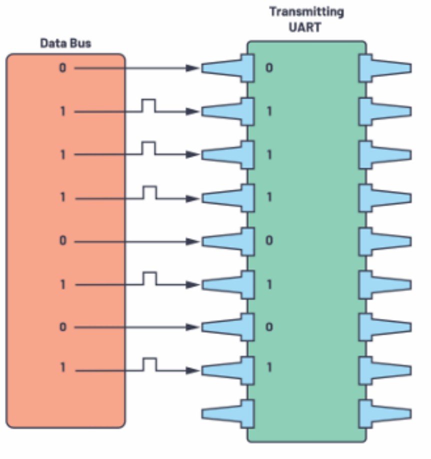


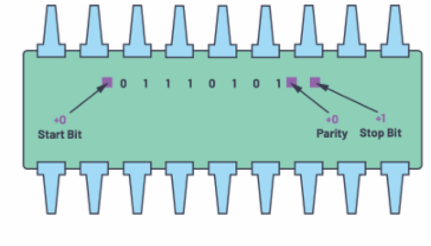
The communication happens in packets which include:

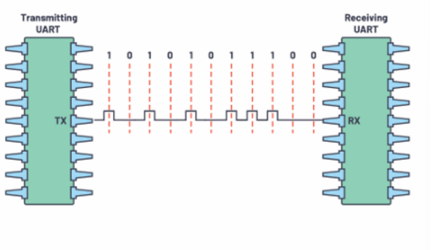
- the start bit, which is identified from a high to low transition, so when the receiver detects a 1→0 transition, it starts sampling the incoming data
- the data frame, which contains the word sent to the UART, and it can have a size from 5 to 8 bits (if parity bit is used) or 9 (if parity bit is not used)

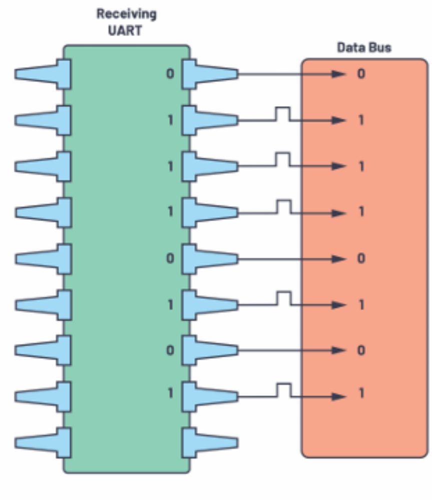
- the parity bit which is used to say if the number of 1s in the data frame is even (parity bit = 0) or odd (parity bit = 1). The receivers counts the number of received 1s and compares it with the parity bit; if they differ one bit changed during transmission, revealing an error. This is good for single bit flips during transmission
- the stop bit which is identified with a low to high transition and it can last 1 or 2 cycles.

UART transmitter receives a parallel word to transmit which corresponds to the data frame. Then, the transmitter attaches start, stop and parity bits and then it communicates with the receiver through a single-bit serial connection. The receiver removes start, stop and parity bits and writes data frame in parallel to the internal bus.









USART

USART is a synchronous protocol where only two transmission lines are used, one for the clock and one for the data, and there is no need for start and stop bits, since receiver and transmitter are synchronized.

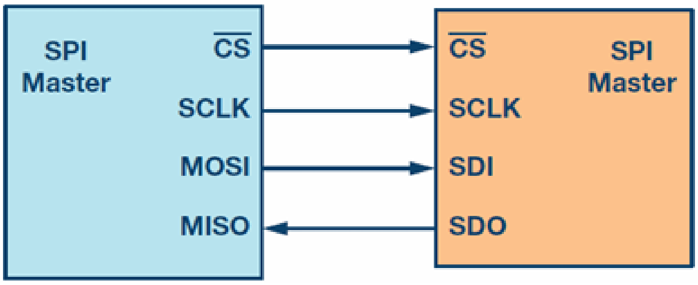
SPI

SPI supports a single master and multiple slaves and communications are clocked (synchronized). It it always full-duplex, meaning it communicates in both directions simultaneously.

We have 4 wires:

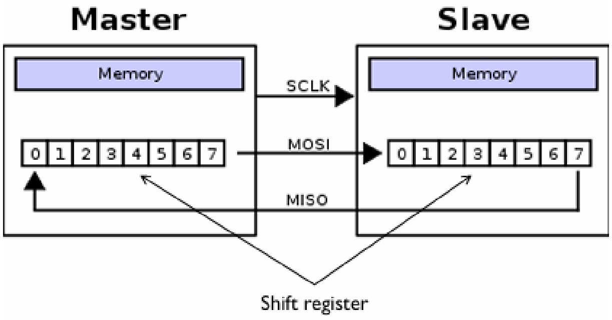
- MOSI, which carries data out of master to slave
- MISO, which carries data out of slave to master
- CS, which is a unique line to select each slave chip
- SCLK, which is produced by master to synchronize transfers

The master asserts slave/chip select line, generates clock signal while both the master and the slave shift data in and out.



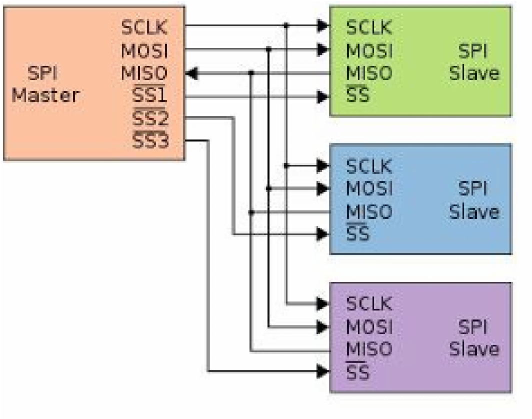
SPI slave to the right*

The master and the slave have one shift register each. At the beginning of the data transfer the shift register contains the data to be sent, while at the end the shift register contains the data received.

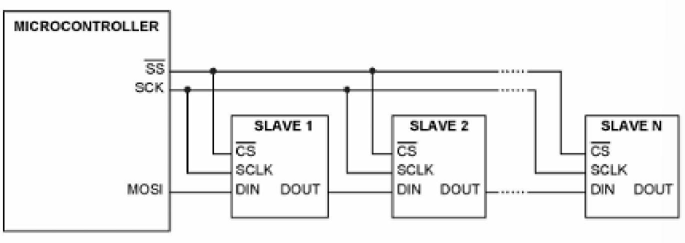


There are different ways to link the master to multiple slaves:

- in a “parallel” way, so that each slave is independent from the others

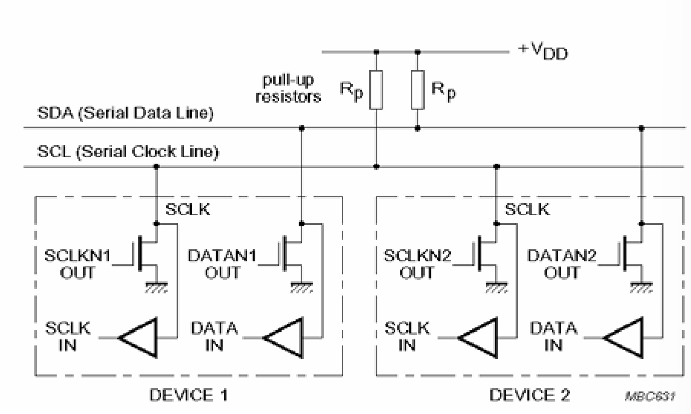


- in a serial way, known as daisy-chained slaves



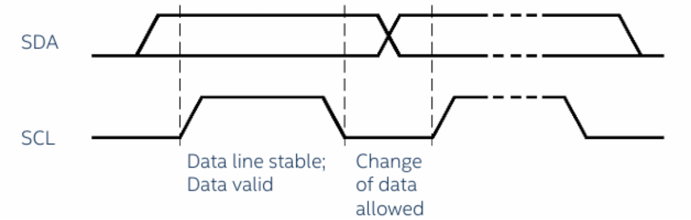
I2C

In inter integrated circuit is a two-wire serial protocol with addressing capability is used, one for serial data (SDA) and one for serial clock (SCL). It allows multi masters and multi slaves, and in it uses pull-up resistors.

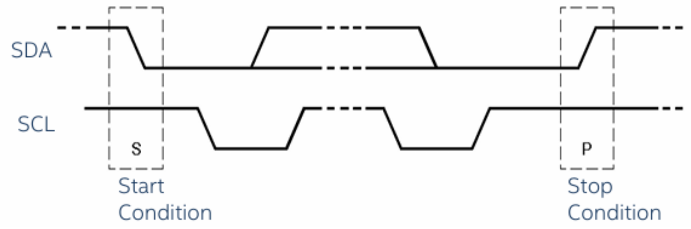


I^2C devices are wire ANDed together
If any single node writes a zero, the entire line is zero

In normal data transfer, the data line only changes state when the clock is low.



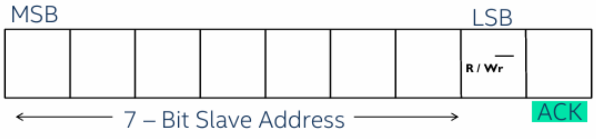
A transition of the data line while the clock line is high is defined as either a start or a stop condition. Both start and stop conditions are generated by the bus master. The bus is considered busy after a start condition, until a stop condition occurs.



ADDRESSING

Each node has a unique 7 (or 10) bit address. Peripherals often have fixed and programmable address portions. Addresses starting with a 0000 or 1111 have special functions:

- 00000000 is a general call address
- 00000001 is a null address
- 1111XXXX is an address extentions
- 11111111 is an address extension where next bytes are the actual address



R/Wr can be:

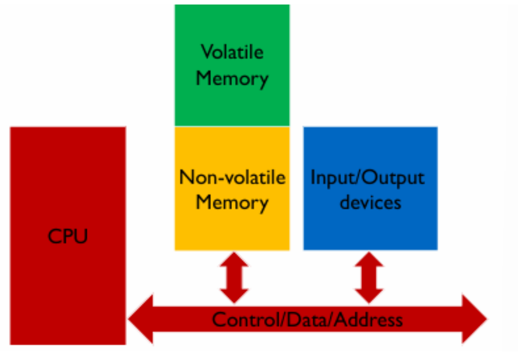
- 0 if the slave is written to by master
- 1 slave read by master

The ACK slave is generated by the slave whose address has been output.

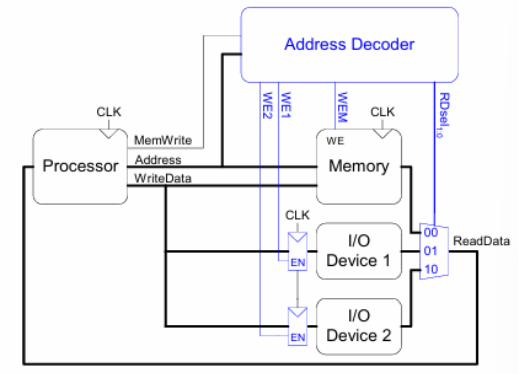
Multi-master situations require additional features of the I2P protocol, such as arbitration, which is the procedure by which competing masters decide final control of the bus; it is performed bit by bit until it is uniquely resolved but it is lost by a master when it attempts to assert a high on the data line and fails.

Memory mapped I/O (MMIO)

In memory mapped I/O, peripheral devices share the same address space of the main memory and there are portions of the address space dedicated to peripherals. It is a flexible solution because to add a new peripheral it is sufficient to map it in a free portion of the address space. It is also easy to access since peripherals are accessed with simple load and store instructions.



Even if peripherals and memory share the same address space, peripherals generally occupy a dedicated portion of it, separated by the main memory. When CPU access an address, the address decoder select the corresponding peripheral and performs the load/store in its specific registers.



I/O interaction

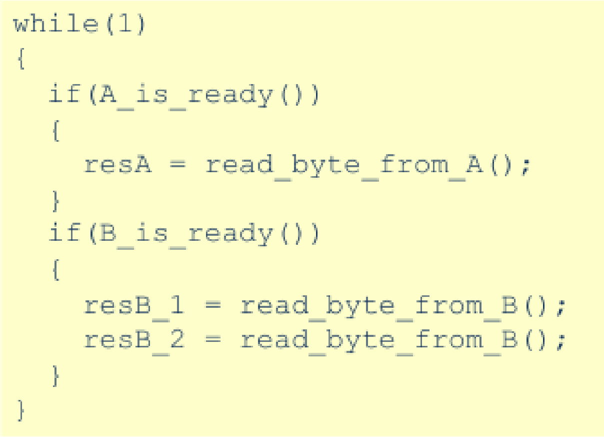
CPU and peripherals operate synchronously; the CPU runs software and the peripheral performs its own tasks. When the CPU has to read or write from or to the peripheral, a read or write cycle is performed, which is always initiated by the CPU.

In order to recognize that a peripheral has a data ready for being read, two techniques are possible:

- polling
- interrupt

Polling

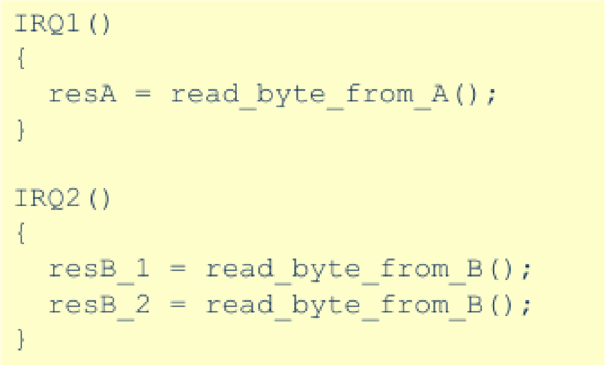
The software periodically reads the status of each peripheral and in case the peripheral needs to be serviced by the CPU, the corresponding program (service routine) is executed. It is simple to implement but it has a high latency as peripherals are polled serially.



```
while(1)
{
  if(A_is_ready())
  {
    resA = read_byte_from_A();
  }
  if(B_is_ready())
  {
    resB_1 = read_byte_from_B();
    resB_2 = read_byte_from_B();
  }
}
```

Interrupt

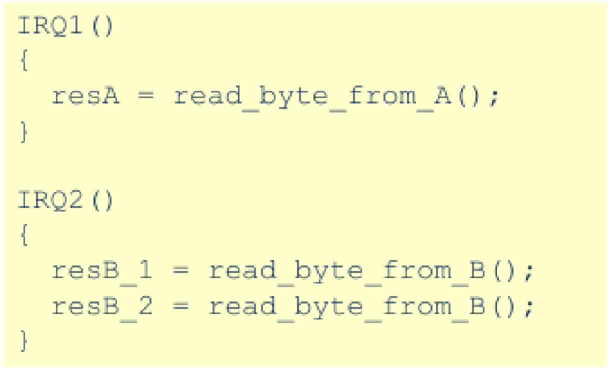
A dedicated line connects the peripherals with the CPU and in case the peripheral needs to be serviced it asserts the interrupt request line. At the end of each instruction execution the CPU checks for the IRQ line, if asserted it runs the corresponding service routine.. It has a low latency but it requires a higher hardware complexity.



```
IRQ1()
{
  resA = read_byte_from_A();
}

IRQ2()
{
  resB_1 = read_byte_from_B();
  resB_2 = read_byte_from_B();
}
```

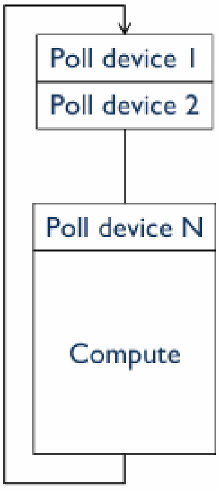
Polling vs interrupt

In case of polling, the application execution timeline will be:



```
IRQ1()
{
  resA = read_byte_from_A();
}

IRQ2()
{
  resB_1 = read_byte_from_B();
  resB_2 = read_byte_from_B();
}
```

while in case of interrupt it will be:



In case of polling, the latency depends on the order in which the peripherals are polled while in case of interrupt, the latency depends on the number of requests simultaneously asserted, on the CPU operating mode and on its architecture.

CPU - Peripheral interface

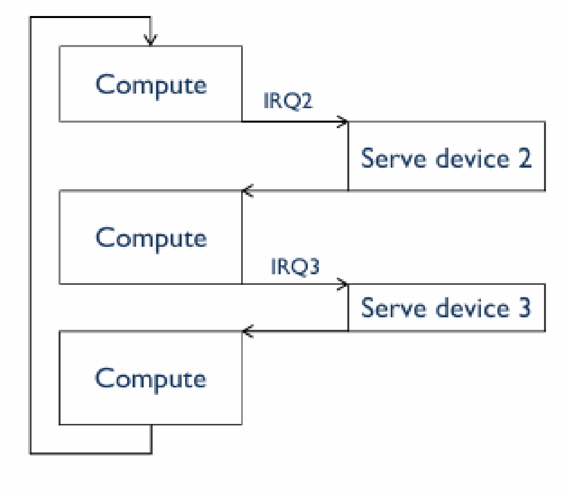
Data transfer can happen in two ways:

- character-based transfer, where data are transferred one word at a time
- block-based transfer, where data are transferred as a cluster of bytes

The data transfer can be performed:

- by the CPU, which moves data from memory to I/O through a program
- by the DMA, which releases the CPU from data transfer

The typical architecture is



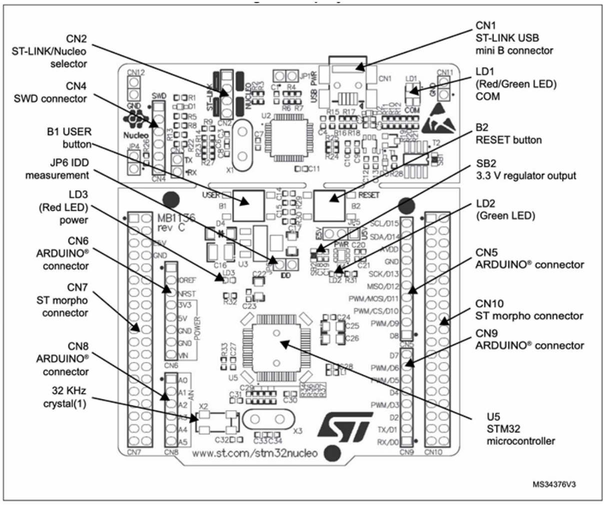
DMA - Direct memory access

The data transfer mode can be:

- burst, in which an entire block of data is transferred in one contiguous sequence and the CPU remains inactive for relatively long periods of time
- cycle stealing, in which DMA transfers one byte of data and then releases the bus returning control to the CPU; this mode continually issues requests, transferring one byte of data per request, until it has transferred the entire block of data; in this case it takes longer to transfer data so the CPU is blocked for less time
- transparent, in which DMA transfers data when the CPU is performing operations that do not use the system buses

Overview of the board

- STM32F446RE
- ARM CORTEX-M4 processor (32 bit)
- 512KB of flash memory
- 128KB of SRAM
- Programmed and debugged through ST-LINK



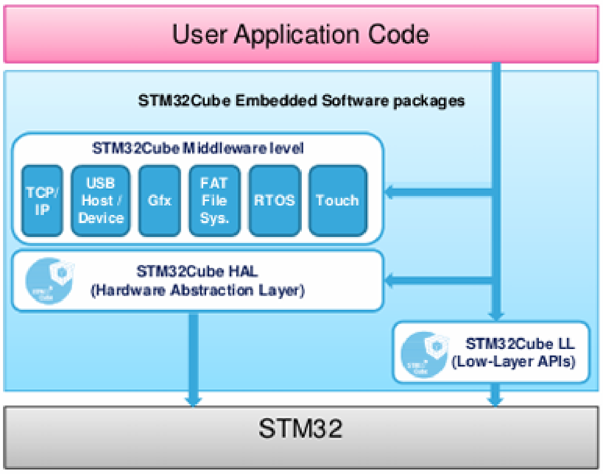
ST-Link

The ST-link is a debugging and programming tool (upper part of the image) and it connects the board to a host computer via USB. It enables programming, debugging and real-time analysis of STM32 devices, based on SWD (Serial wire debug).

SWD is a two-pin protocol, SWDIO for data and SWCLK for clock, and it is designed for ARM Cortex-M microcontrollers. It provides a low-overhead alternative to the traditional JTAG interface and it is used by ST-link to communicate with STM32 microncontrollers.

Hardware abstraction layer (HAL)

A HAL is used when large amount of memory-mapped peripherals on the board and users must read and understand the user manual and check it everytime they want to access a peripheral. It simplifies access to the peripherals which is performed with high-level API which performs the low-level access seen before.


User Application Code
STM32Cube Embedded Software packages
STM32Cube Middleware level
TCP/IP
USB Host / Device
Gfx
FAT File Sys.
RTOS
Touch
STM32Cube HAL
(Hardware Abstraction Layer)
STM32Cube LL
(Low-Layer APIs)
STM32