Mastering Microcontroller : TIMERS, PWM, CAN, RTC,LOW POWER

FastBit Embedded Brain Academy
Check all online courses at
www.fastbitlab.com

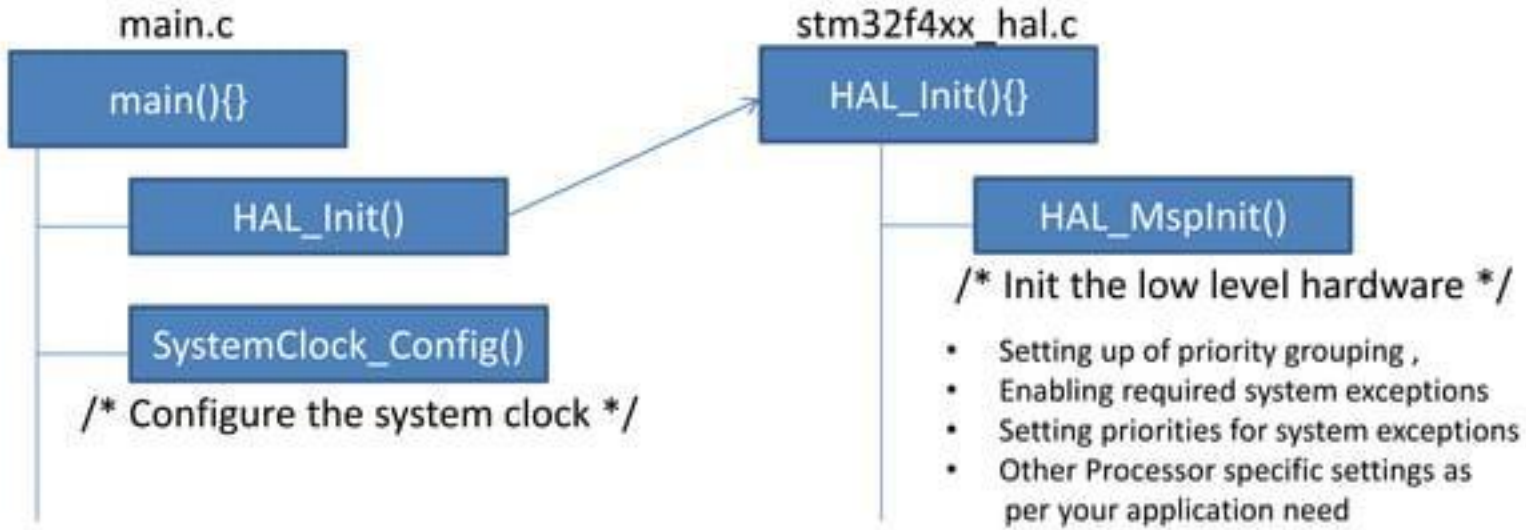
About FastBit EBA

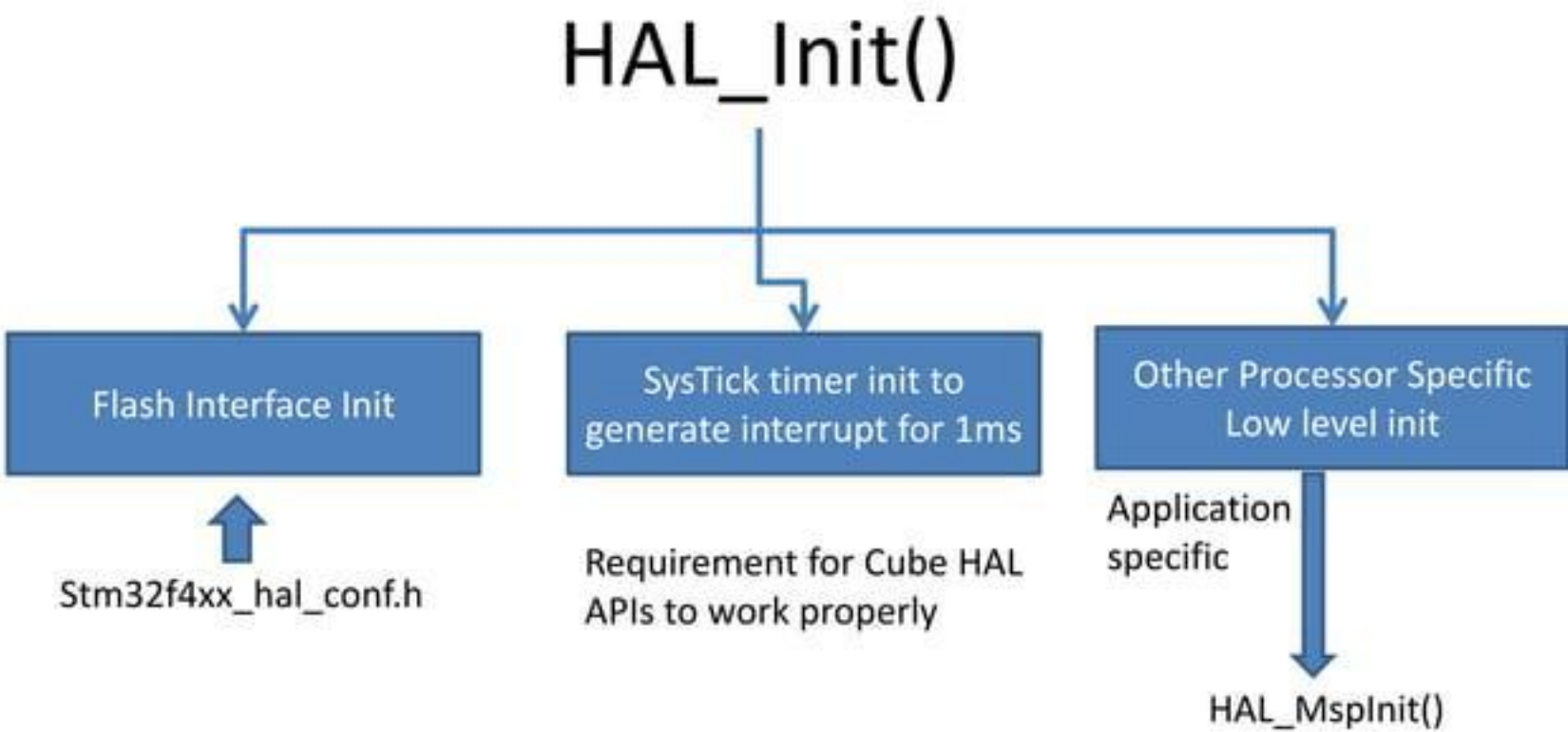
FastBit Embedded Brain Academy is an online training wing of Bharati Software. We leverage the power of internet to bring online courses at your fingertip in the domain of embedded systems and programming, microcontrollers, real-time operating systems, firmware development, Embedded Linux.

All our online video courses are hosted in Udemy E-learning platform which enables you to
exercise 30 days no questions asked money back guarantee.

For more information please visit : www.fastbitlab.com
Email : contact@fastbitlab.com

STM32 Cube Framework Program Flow

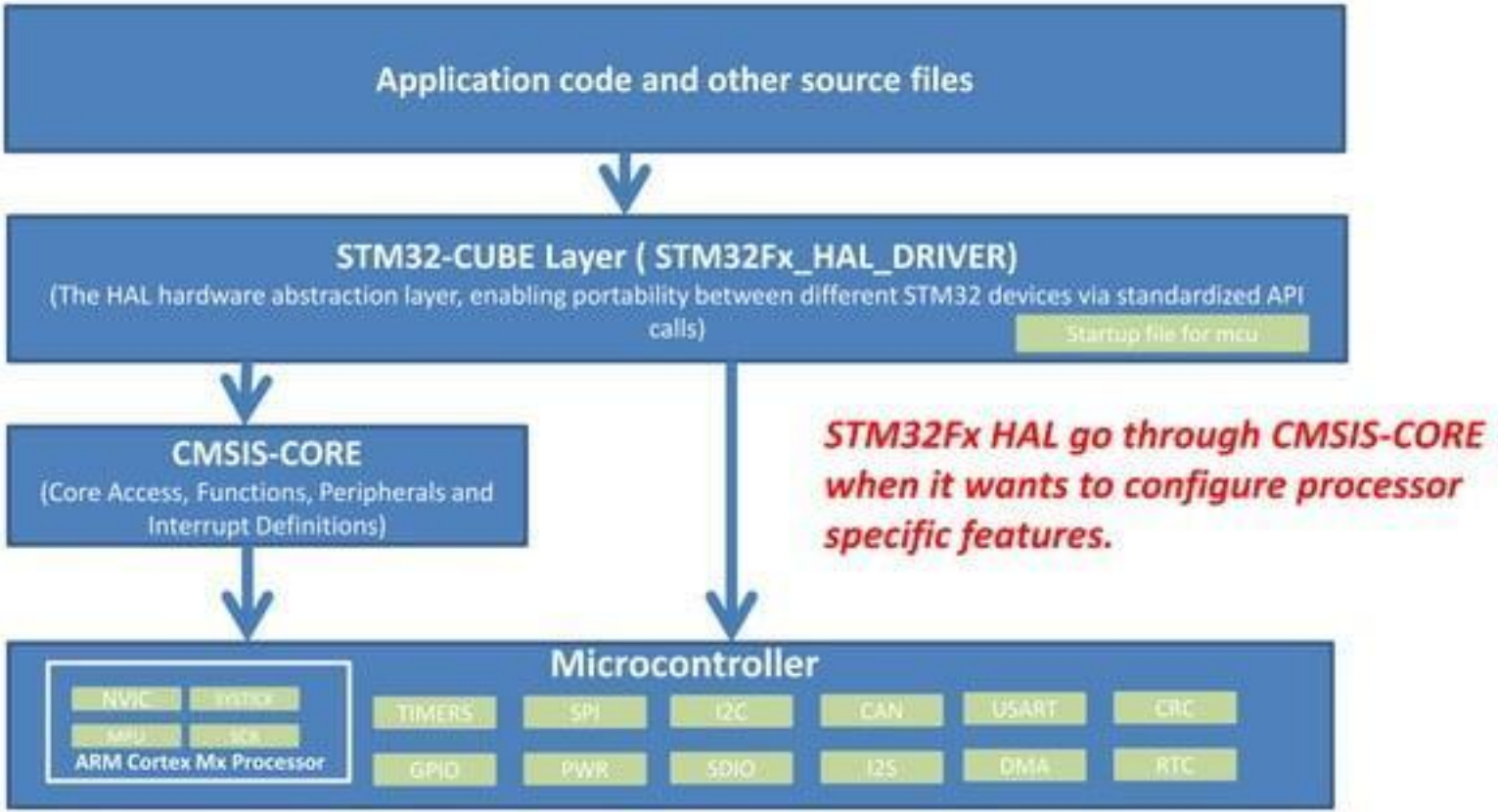


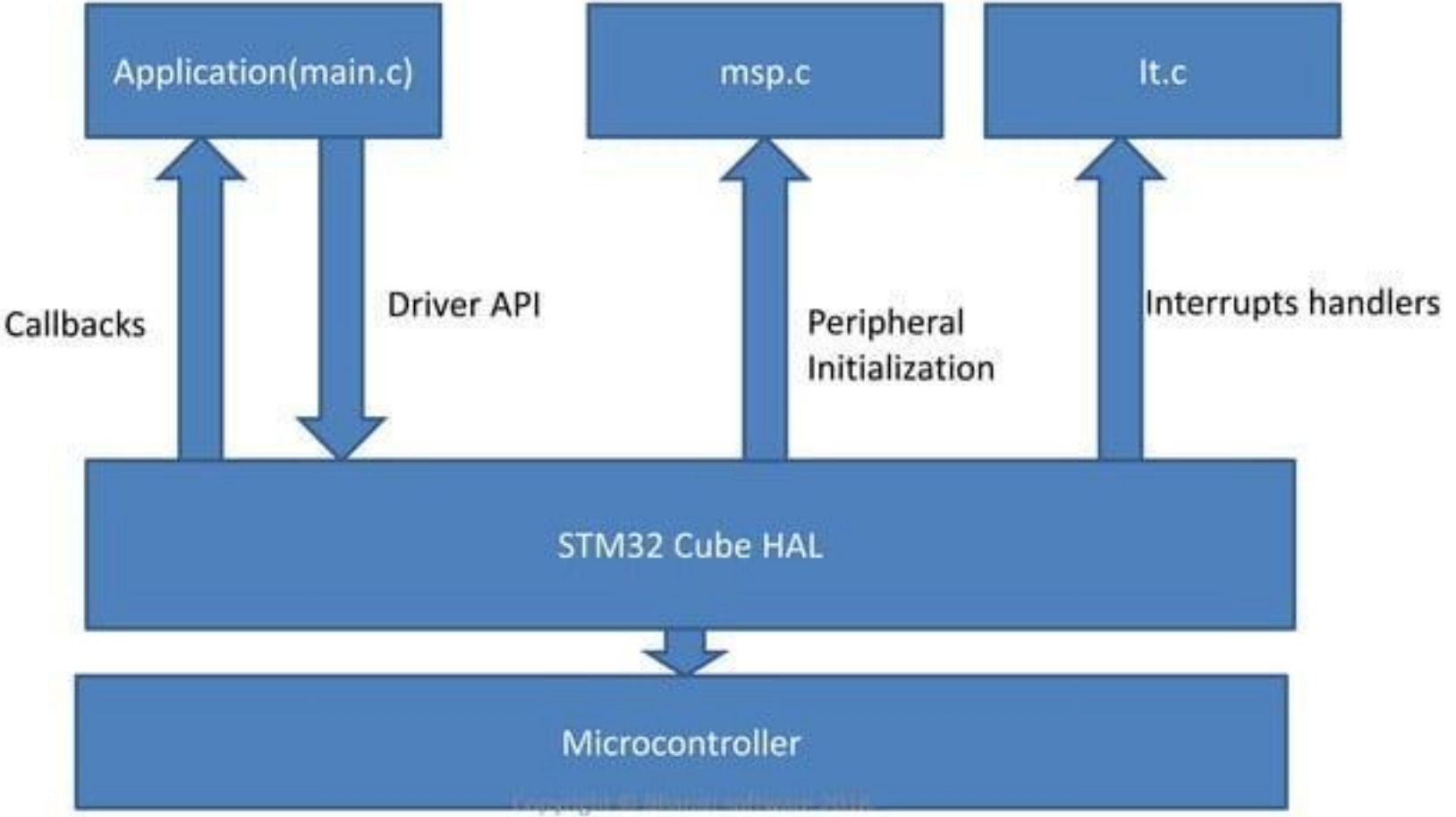


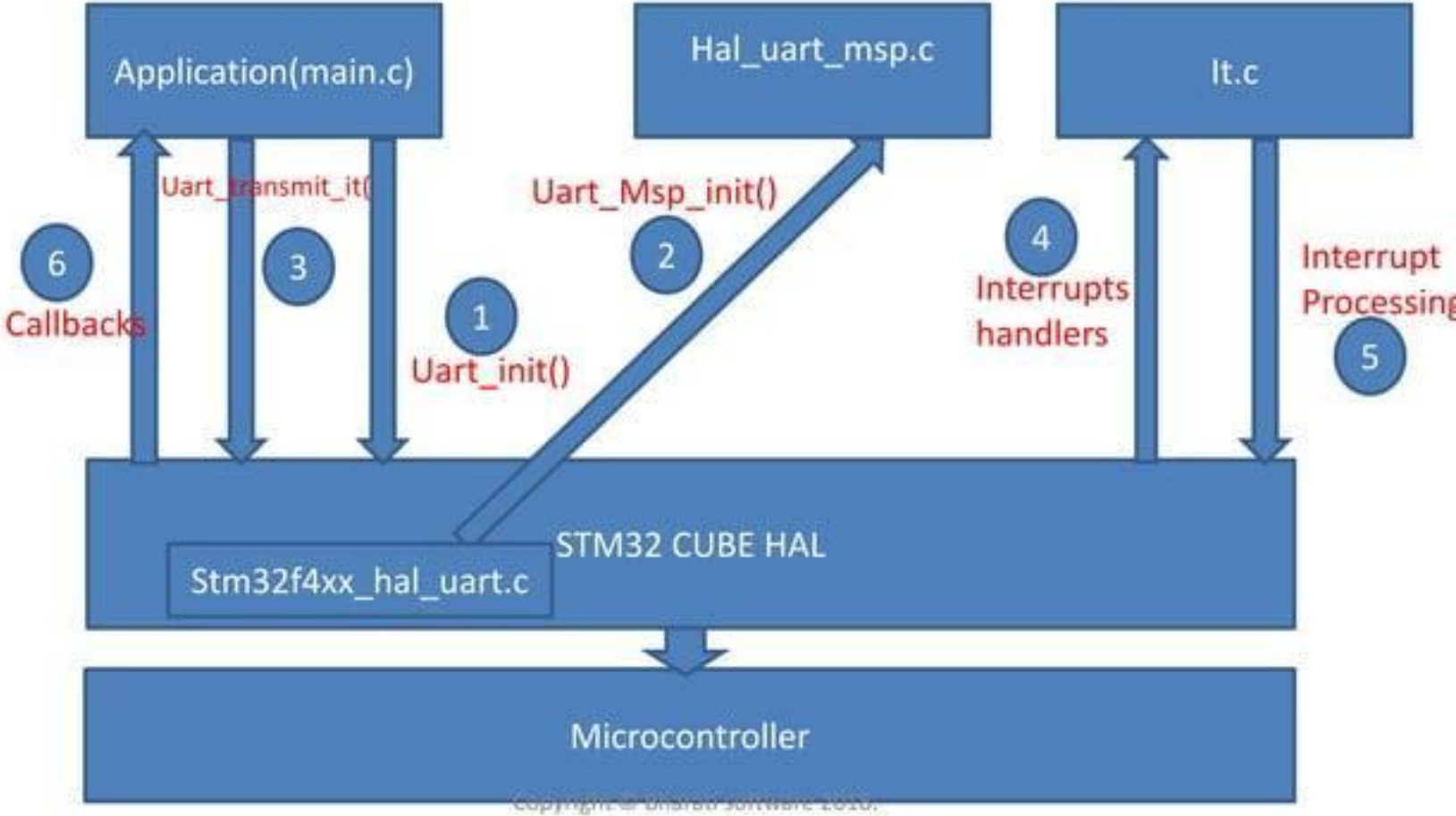
SysTick Init

SysTick is used as time base for the HAL_Delay() function, the application

need to ensure that the SysTick time base is always set to 1 millisecond to have correct HAL operation.




Application(main.c)
msp.c
It.c
Callbacks
Driver API
Peripheral Initialization
Interrupts handlers
STM32 Cube HAL
Microcontroller


Application(main.c)
Hal_uart_msp.c
It.c
Uart_transmit_it(
Uart_Msp_init()
6
Callbacks
3
2
1
Uart_init()
4
Interrupts
handlers
Interrupt
Processing
5
STM32 CUBE HAL
Stm32f4xx_hal_uart.c
Microcontroller

Peripheral Handle Structure

Garden Spade With Handle

Handle

Spade

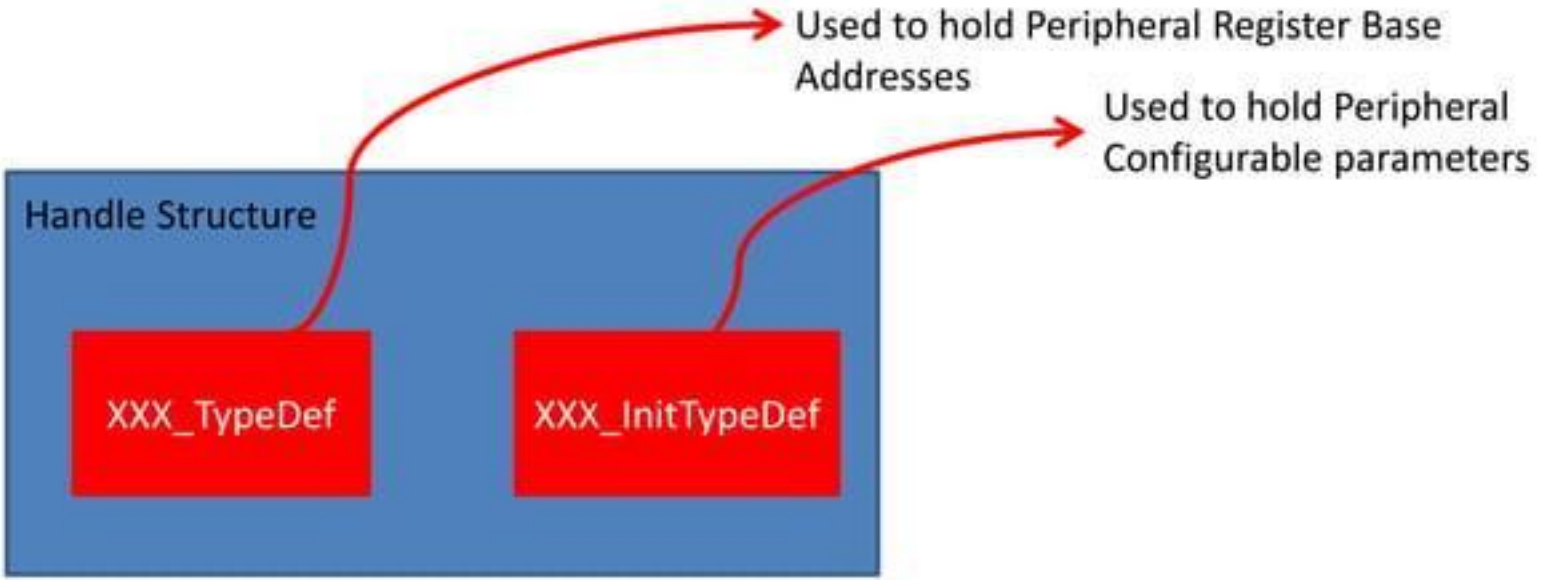
UART_HandleTypeDef huart2;

UART peripheral of the MCU

UART Handle Structure

```
/**
  * @brief  UART handle Structure definition
  */
typedef struct
{
  USART_TypeDef                 *Instance;        /*!< UART registers base address        */

  UART_InitTypeDef              Init;             /*!< UART communication parameters      */

  uint8_t                       *pTxBuffPtr;      /*!< Pointer to UART Tx transfer Buffer */

  uint16_t                      TxXferSize;       /*!< UART Tx Transfer size              */

  __IO uint16_t                 TxXferCount;      /*!< UART Tx Transfer Counter           */


  __IO uint32_t                 ErrorCode;        /*!< UART Error code                    */

}UART_HandleTypeDef;
/**
```

Peripheral Handle Structure



Handle Structure of different Peripherals in STM32 Cube

Peripheral	Handle Structure	Driver file
Timer	TIM_HandleTypeDef	stm32f4xx_hal_tim.h
RTC	RTC_HandleTypeDef	stm32f4xx_hal_rtc.h
SPI	SPI_HandleTypeDef	stm32f4xx_hal_spi.h
I2C	I2C_HandleTypeDef	stm32f4xx_hal_i2c.h
USART	UART_HandleTypeDef	stm32f4xx_hal_uart.h
CAN	CAN_HandleTypeDef	stm32f4xx_hal_can.h
RCC		

Handle Structure of different Peripherals

Peripheral	Handle Structure	
DMA	DMA_HandleTypeDef	stm32f4xx_hal_dma.h
SDIO	SD_HandleTypeDef	stm32f4xx_hal_sd.h
GPIO		
MII	ETH_HandleTypeDef	stm32f4xx_hal_eth.h
FLASH	FLASH_ProcessTypeDef	stm32f4xx_hal_flash_ex.h
ADC	ADC_HandleTypeDef	stm32f4xx_hal_adc.h
CRC	CRC_HandleTypeDef	stm32f4xx_hal_crc.h

XXX_TypeDef

This structure you will find in device specific header file

For example, in the case of STM32F446RE MCU, it is stm32f446xx.h

This structure describes the register details of a particular peripheral in the order they appear in the memory map.

XXX_TypeDef: Example

```
/**
  * @brief Universal Synchronous Asynchronous Receiver Transmitter
  */

typedef struct
{
  __IO uint32_t SR;         /*!< USART Status register,                   Address offset: 0x00 */
  __IO uint32_t DR;         /*!< USART Data register,                     Address offset: 0x04 */
  __IO uint32_t BRR;        /*!< USART Baud rate register,                Address offset: 0x08 */
  __IO uint32_t CR1;        /*!< USART Control register 1,                Address offset: 0x0C */
  __IO uint32_t CR2;        /*!< USART Control register 2,                Address offset: 0x10 */
  __IO uint32_t CR3;        /*!< USART Control register 3,                Address offset: 0x14 */
  __IO uint32_t GTPR;       /*!< USART Guard time and prescaler register, Address offset: 0x18 */
} USART_TypeDef;
```

Linking Handler Structure and Peripheral

Linking happens by means of base address of the peripheral.

```
/**
  * @brief  UART handle Structure definition
  */
typedef struct
{
  USART_TypeDef                 *Instance;        /*!< UART registers base address        */

  UART_InitTypeDef              Init;             /*!< UART communication parameters      */

  uint8_t                       *pTxBuffPtr;      /*!< Pointer to UART Tx transfer Buffer */

  uint16_t                      TxXferSize;       /*!< UART Tx Transfer size              */

  __IO uint16_t                 TxXferCount;      /*!< UART Tx Transfer Counter           */

  __IO uint32_t                 ErrorCode;        /*!< UART Error code                    */
}UART_HandleTypeDef;
```

How to find out the base address of the peripheral ??

1. Find out on which bus the peripheral is connected
2. Find out the base address of the Bus on which the peripheral is connected. (Reference manual)
3. Find out the offset of the peripheral from the base address of the bus on which it is connected
4. Add “offset” to base address of the bus

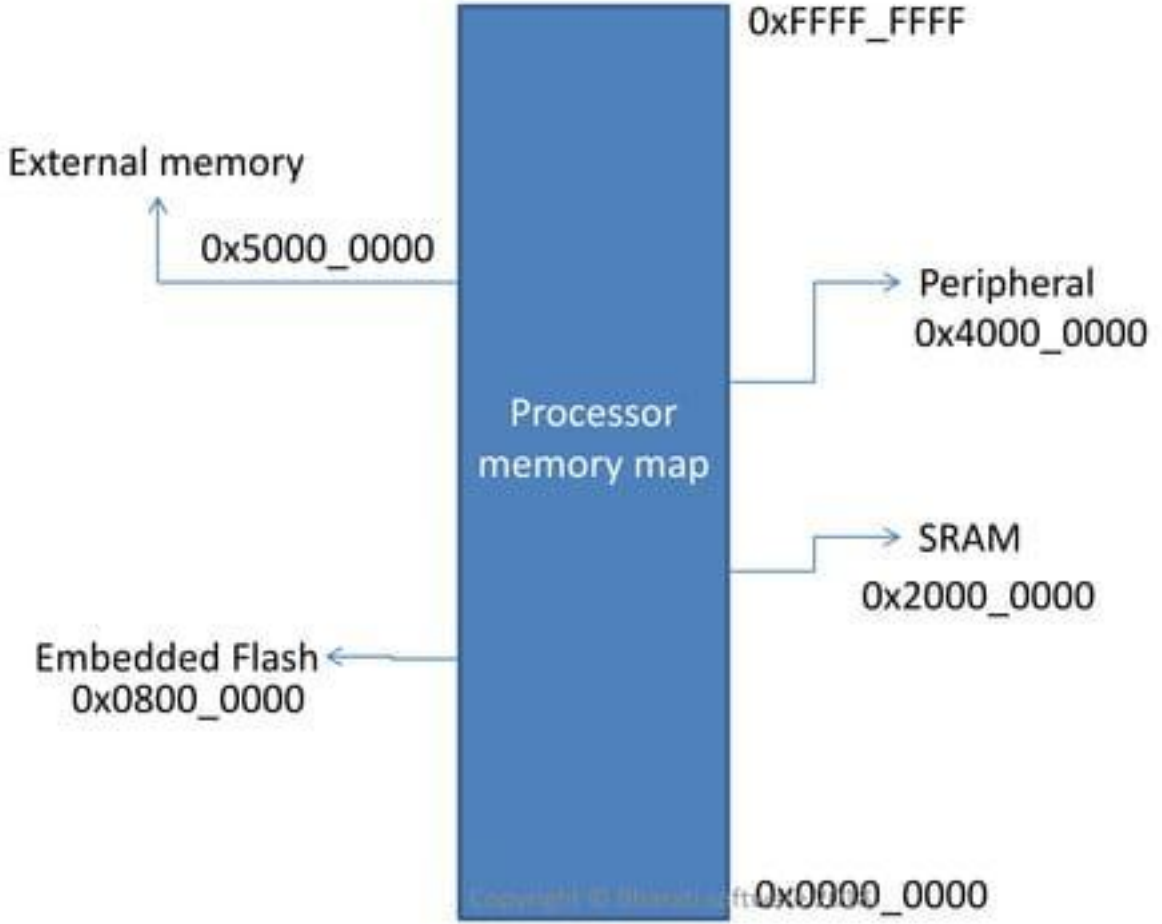
Address calculation : Example

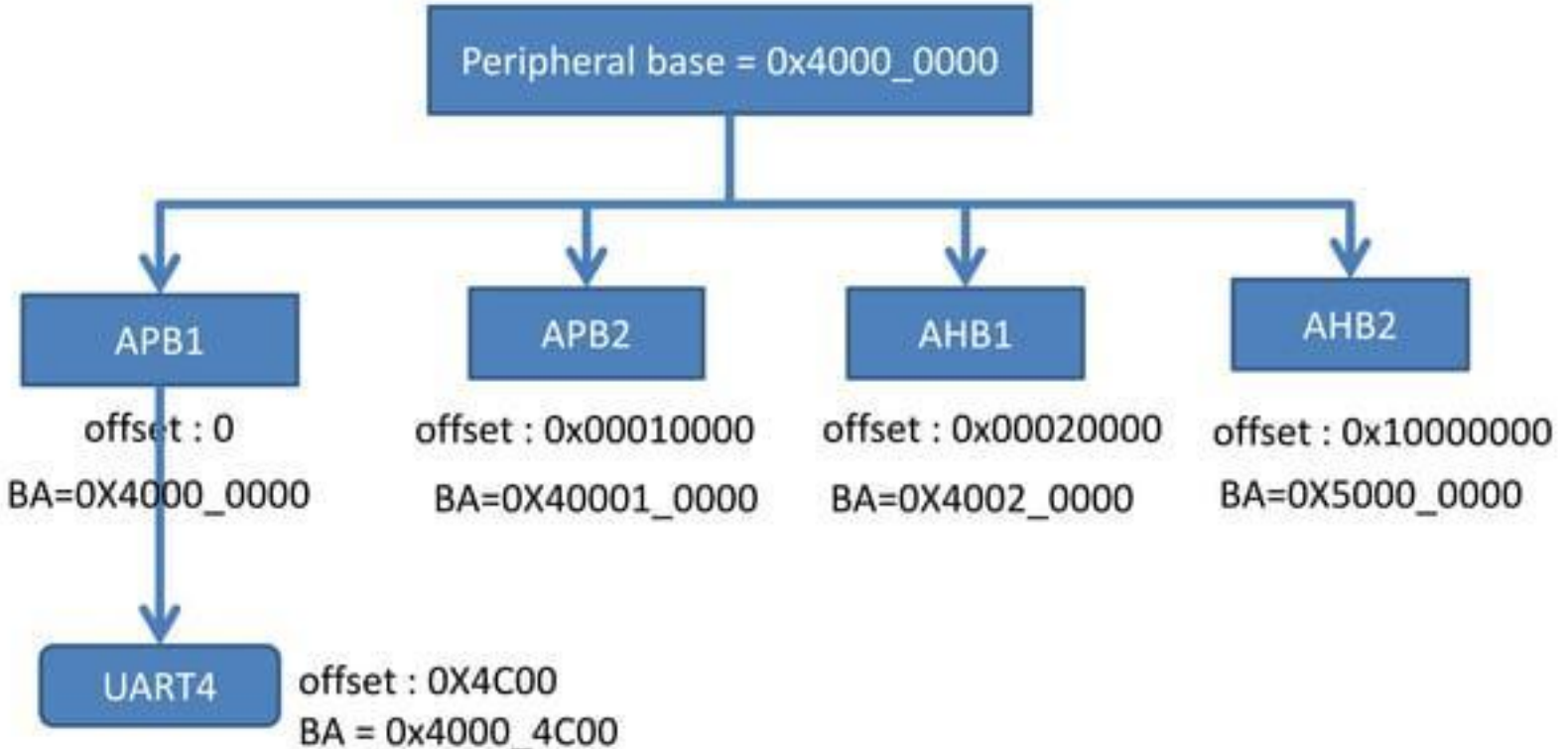
Find out the base address of the UART4 peripheral

1. Refer to the microcontroller datasheet or RM to find out on which bus it is connected

Address calculation : Example

2. *Find out the base address of the bus in the processor memory map*

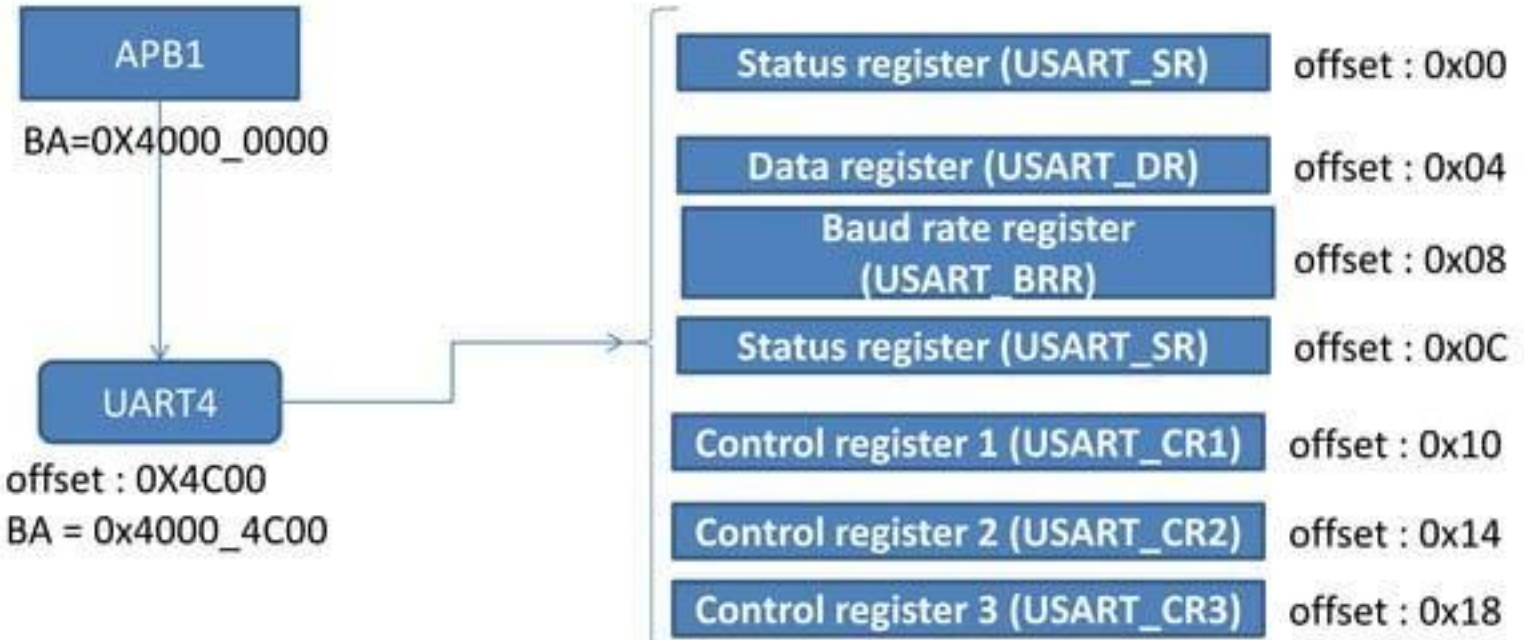

0xFFFF_FFFF
External memory
0x5000_0000
Peripheral
0x4000_0000
Processor
memory map
SRAM
0x2000_0000
Embedded Flash
0x0800_0000
0x0000_0000



Address calculation : Example

3. Find out the offset of the peripheral from the base address of the bus on which it is connected

Peripheral Register Address Calculation



Linking Handler Structure and Peripheral

```
/**
  * @brief  UART handle Structure definition
  */
typedef struct
{
  USART_TypeDef                 *Instance;        /*!< UART registers base address        */
  UART_InitTypeDef              Init;             /*!< UART communication parameters      */
  uint8_t                       *pTxBuffPtr;      /*!< Pointer to UART Tx transfer Buffer */
  uint16_t                      TxXferSize;       /*!< UART Tx Transfer size              */
  __IO uint16_t                 TxXferCount;      /*!< UART Tx Transfer Counter           */
  __IO uint32_t                 ErrorCode;        /*!< UART Error code                    */
}UART_HandleTypeDef;
```

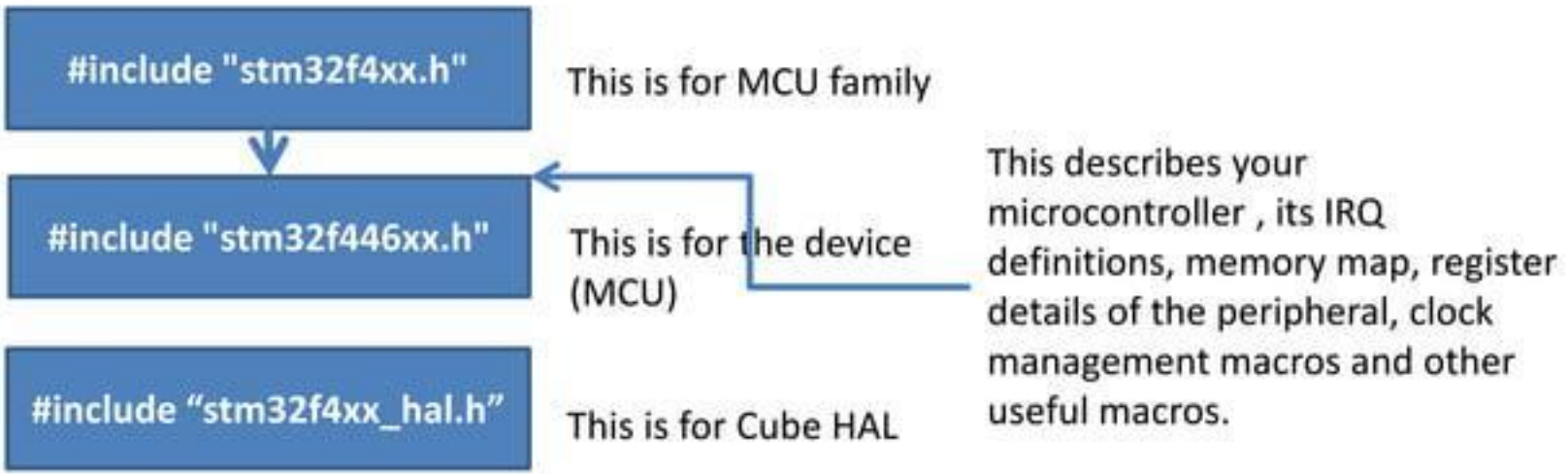
Handle

Spade

UART_HandleTypeDef huart2;

UART peripheral of the MCU

STM32 Cube Header File Hierarchy



Exercise

Write a program to send characters over UART to MCU . MCU should convert all the lower case letters in to uppercase and send it back to the user

Consider the below UART Parameter settings

Baudrate = 115200

No of data bits = 8

Stop bits = 1

Parity = None

No UART hardware flow control.

Use USART2 peripheral of the MCU in Asynchronous mode(i.e UART mode)

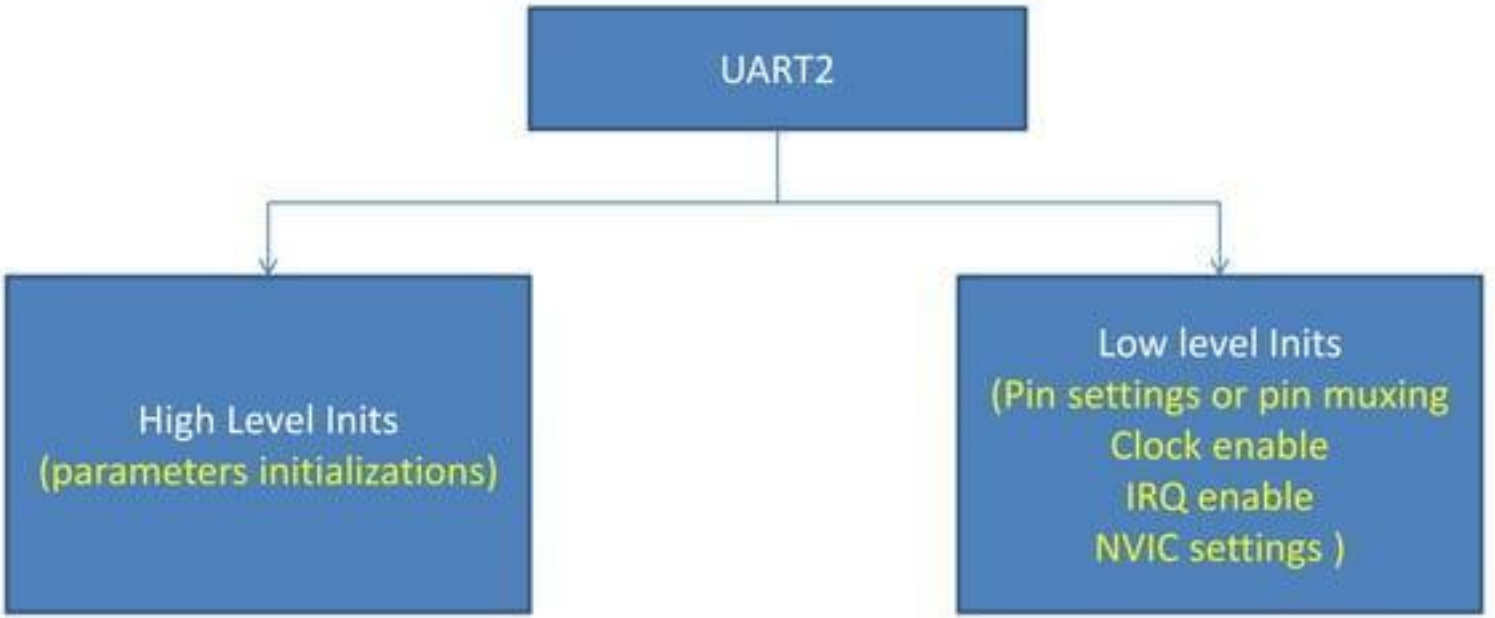
Low Level Hardware Initializations

- Processor Specific
  - Configure the Priority group of the processor
  - Enable required Processor System Exceptions
    - Bus fault , mem manage, usage fault , Systick , etc
  - Configure the priority of system exceptions
  - Other Initializations related to MPU, Floating point unit, Sleep mode , etc

Peripheral Initializations : UART2

- First define a handle variable to Handle the required peripheral as a global variable in the program
- Link as well as initialize the handle variable
- Use the init. API of the Peripheral to initialize the peripheral .

Low level initializations of the peripheral



Low level initializations of the peripheral

- First enable the Required Peripheral clock
- Do pin muxing configurations
- Enable the peripheral IRQ in the NVIC
- Set the priority as per your application need.

Pin Muxing Configurations

- UART needs 2 pins for Communication . One is called Tx Pin and another one is called Rx Pin
- We can use 2 GPIOs of the MCU for TX and RX functionalities of the UART.
- Before using GPIOs for UART communication purpose , we have to configure their mode as alternate functionality mode.
- In this case alternate functionality mode is UART.

Identifying Pin Packs

Identify all Pin Packs of the MCU on which USART2 communicates.

Hint : Refer to the datasheet of the MCU

Summary

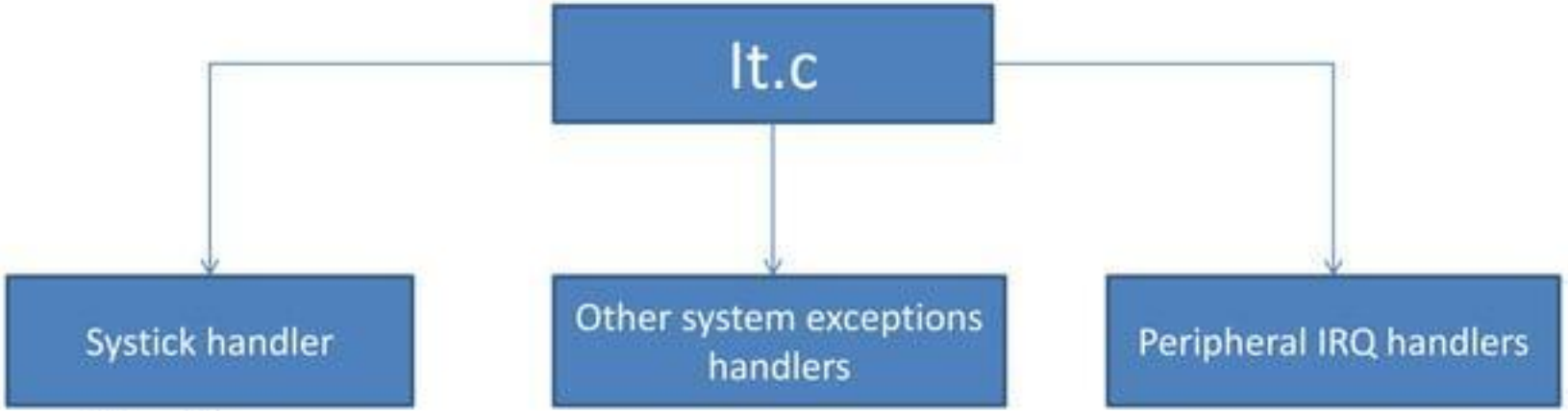
- HAL_Init(); must be the very first API should be called in a project which uses stm32 cube HAL. This call also sets up systick timer ticking for every 1 ms which is the heart beat of stm32 cube project (mandatory)
- HAL_Init(); calls HAL_MspInit(); to do application specific low level processor inits. (optional)
- At this time you are good to go since most of the basic Inits are done

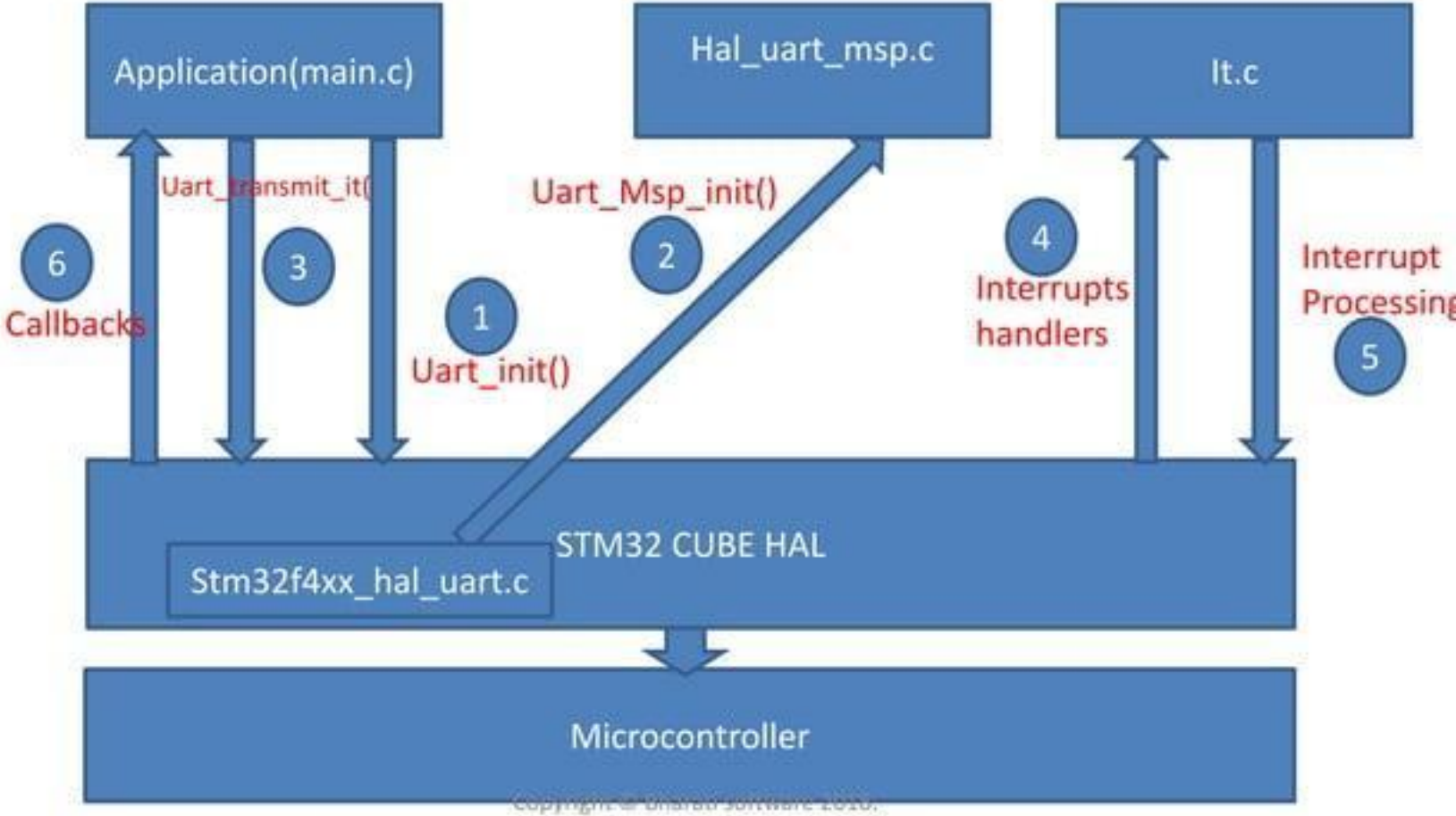
Summary

- SystemClock_Config() has to be called next if you have any special clock requirements as per you application .
- Then we did , UART peripheral initialization (high level inits). That is the parameter settings of the UART peripheral
- Then the cube framework calls for low level peripheral inits. , which we did in the msp.c . like enabling the peripheral clock, pin mux configuration and NVIC settings.
- After this, both your processor and microcontroller peripheral is good to go. Use data handling APIs to play with the peripheral .

Handing interrupts in the project

- Populate the it.c file with required IRQ handlers definitions
- The name of the IRQ handler you can get from the startup file
- STM32 Cube Framework gives IRQ processing API for every peripheral . For example : it gives HAL_UART_IRQHandler() to process UART global interrupt, which must be called from the IRQ handler.
- The IRQ handler then call the callbacks to the user application.




Application(main.c)
Hal_uart_msp.c
It.c
Uart_transmit_it(
Uart_Msp_init()
6
Callbacks
3
2
1
Uart_init()
4
Interrupts
handlers
Interrupt
Processing
5
STM32 CUBE HAL
Stm32f4xx_hal_uart.c
Microcontroller

Stm32 Cube Peripheral Data Handling API Flavors

- Non Interrupt based (Polling)
- Interrupt based
- DMA based (Uses DMA and Interrupts)

Timers

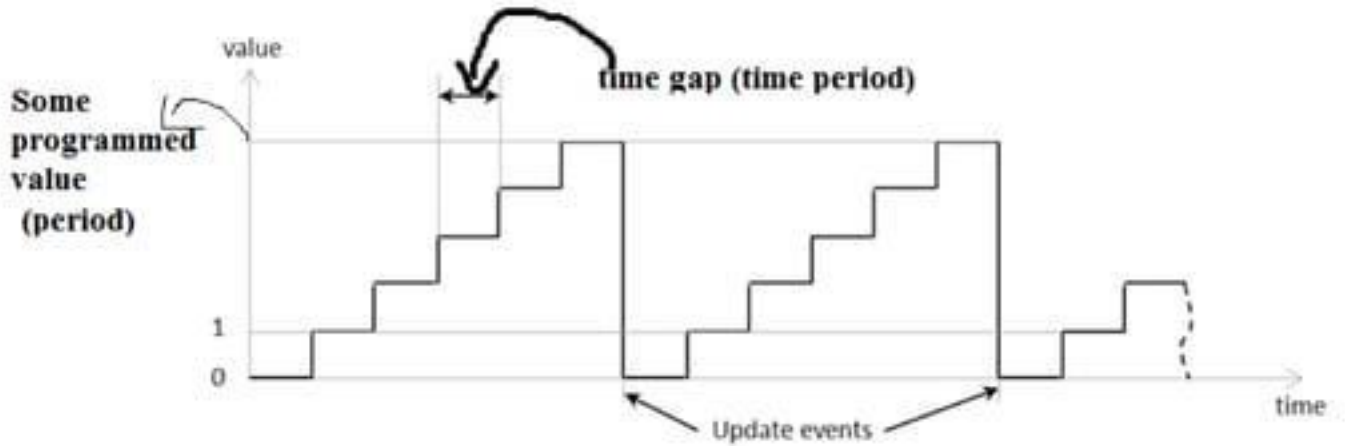
- What exactly is a Timer ?
  - Its one of the peripheral of the Microcontroller .

- What are the uses of a Timer ?
  - Time base generation
  - Measuring/counting input signal frequency (measuring time periods of waveforms)
  - Producing different Waveforms
  - Measuring pules width
  - Generating pulse width modulation (PWM) signals
  - triggering external devices

Timers

- What does a timer do ?
  - A timer at the basic level just counts from 0 to some pre-programmed value (up counting) or from some pre programmed value to 0 (down counting)

Job of the Timer Peripheral is to count



Types of STM32 Timers

- Basic Timers (Available in All STM32 MCUs)
- General Purpose timer (Available in All STM32 MCUs)
- Advanced Timer (not available in all STM32 MCUs)

Timer availability

- *Refer to the timer availability table in application note*

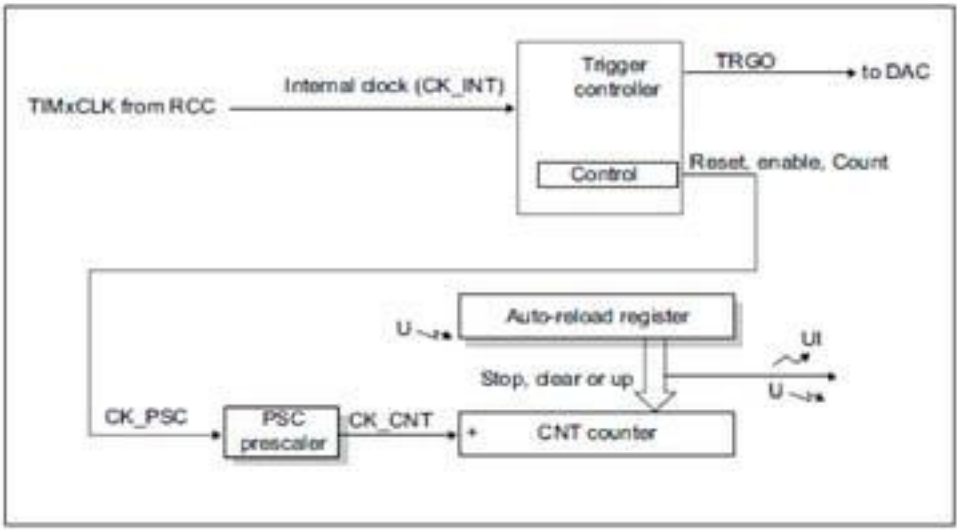
Summary

- The number of timer peripherals and their respective features differ from one STM32 microcontroller family to another, but they all share some common features and operating modes.
- For example, the STM32F100 microcontrollers embed a timer peripheral named TIM17, but the total number of timer peripherals embedded by these microcontrollers is less than 17
- In general, across the STM32 microcontrollers families, the timer peripherals that have the same name also have the same features set, but there are a few exceptions
- The level of features integration for a given timer peripheral is decided based on the applications field that it targets.

Basic Timers

- These timers have basic counting engine and majorly used for time base generation
- These timers do not have input/output channels associated with them

Stm32 Basic Timer Assembly (Block Diagram)



Time base unit

Time base unit includes,

1. 16-bit upcounter
2. Counter Register (TIMx_CNT)
3. Prescaler Register (TIMx_PSC)
4. Auto-Reload Register (TIMx_ARR)

We need to understand .....

- TIM_CLK ?? → Timer Clock
- CLK_PSC ?? → Prescaler output clock
- CLK_CNT ??→ Counter Clock
- ARR ?? → Auto Reload register

Exercise

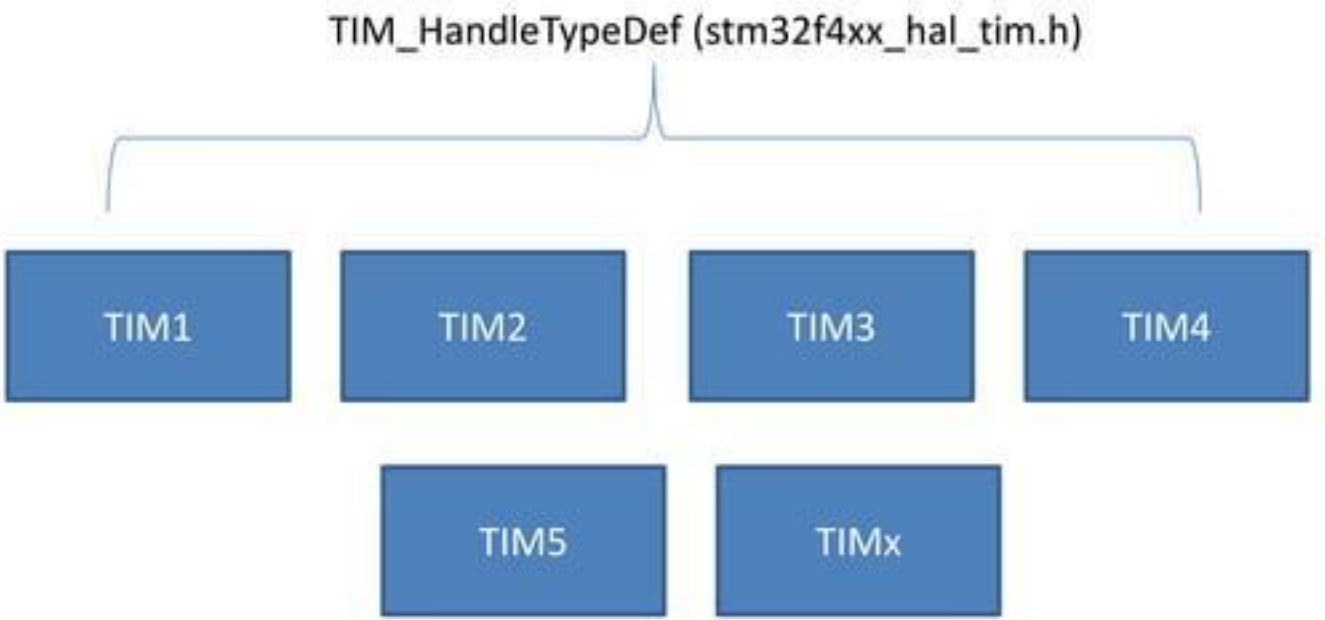
Use the basic timer to generate interrupt for every 100 ms. Toggle the GPIO or LED inside the Timer IRQ handler and verify using logic analyzer

We need more info ….

- What is the peripheral bus clock to which the timer peripheral is connected ?
- Use peripheral bus clock to deduce the Timer clock(TIM_CLK) frequency

We at least need to know the Timer Clock frequency to generate desired time base

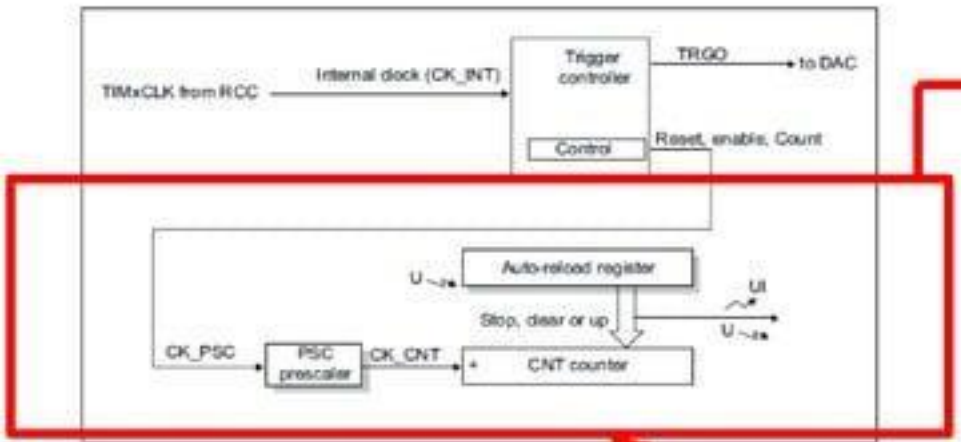
Time base Handle Structure



Time base Handle Structure

```
/**
  * @brief  TIM Time Base Handle Structure definition
  */
typedef struct
{
  TIM_TypeDef              *Instance;     /*!< Register base address              */
  TIM_Base_InitTypeDef     Init;          /*!< TIM Time Base required parameters  */
  HAL_TIM_ActiveChannel    Channel;       /*!< Active channel                     */
  DMA_HandleTypeDef        *hdma[7];      /*!< DMA Handlers array
                                           This array is accessed by a @ref DMA_Handle_i
  HAL_LockTypeDef          Lock;          /*!< Locking object                     */
  __IO HAL_TIM_StateTypeDef State;        /*!< TIM operation state                */
}TIM_HandleTypeDef;
/**
```

Timers and STM32 Cube Layer




```
typedef struct
{
  TIM_TypeDef                 *Instance;
  TIM_Base_InitTypeDef        Init;
  HAL_TIM_ActiveChannel       Channel;
  DMA_HandleTypeDef           *hdma[7];

  HAL_LockTypeDef             Lock;
  __IO HAL_TIM_StateTypeDef   State;
}TIM_HandleTypeDef;
```

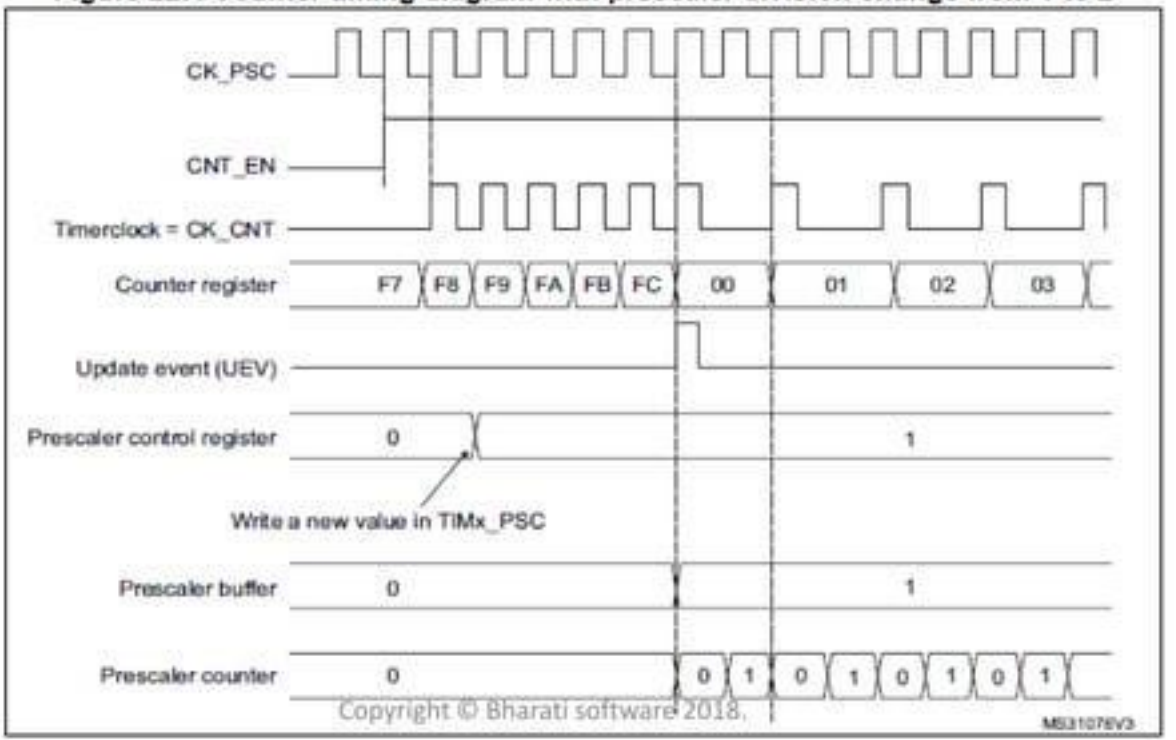
Time base unit of the Timer

How prescaler affects TIMx_CLK??

$$\text{prescaler out put } (CNT\_CLK) = \frac{TIMx\_CLK}{(prescaler + 1)}$$

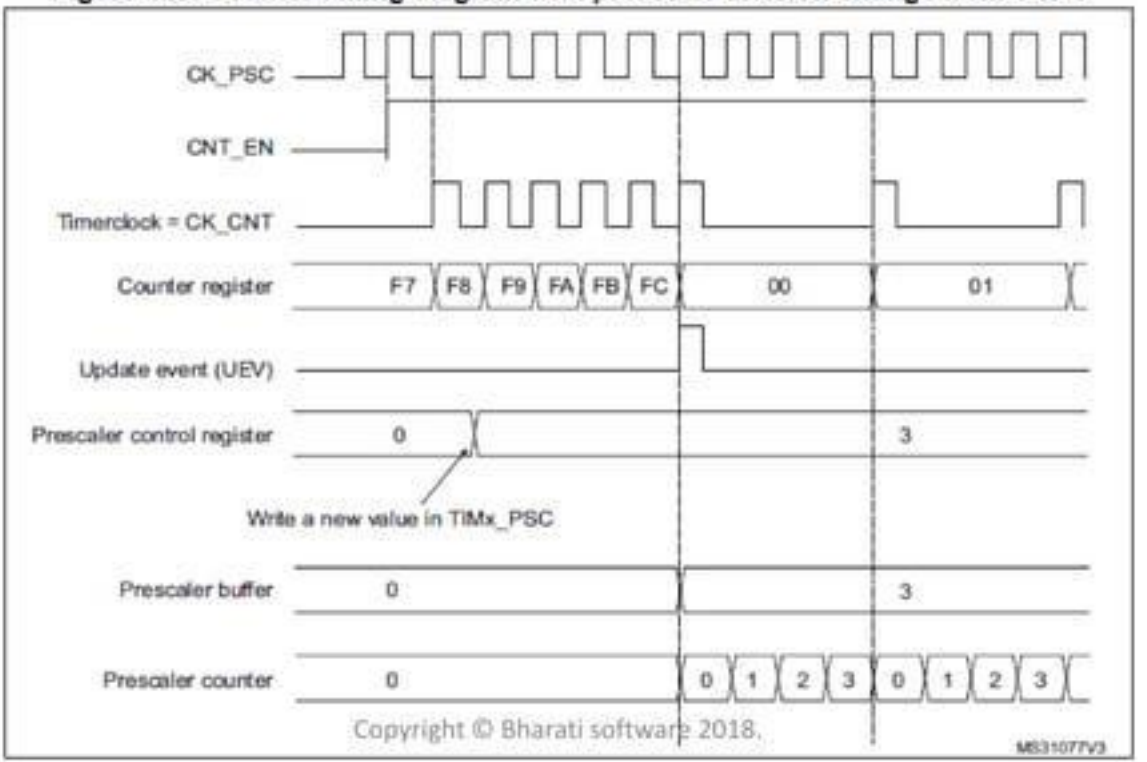
How pre-scaler affects counting ??

Figure 227. Counter timing diagram with prescaler division change from 1 to 2

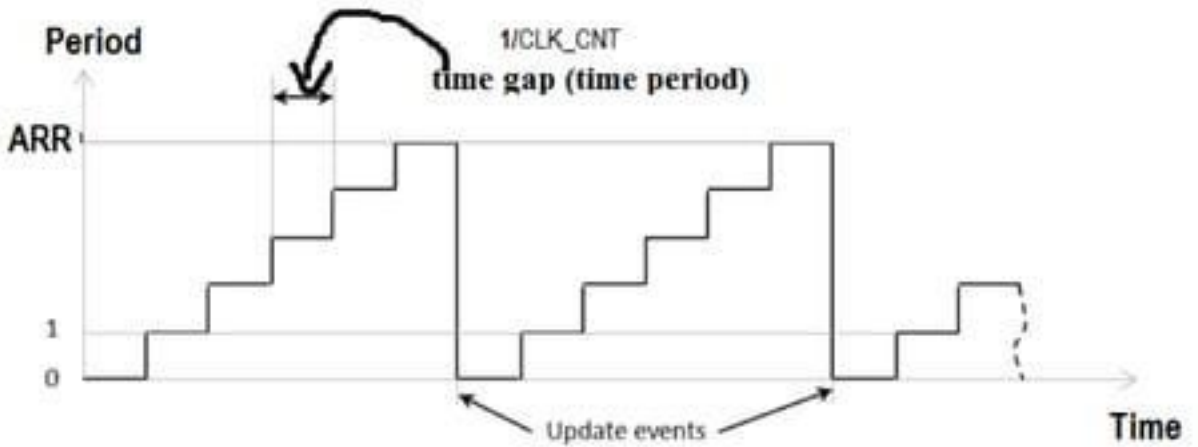


How prescaler affects counting ??

Figure 228. Counter timing diagram with prescaler division change from 1 to 4



Period (ARR value)



Time base generation formula

$$\text{Update\_event} = \frac{\text{TIM\_CLK}}{((\text{PSC} + 1)*(\text{ARR} + 1))}$$

The Time-Base unit

- The time-base unit is made by the timer counter in addition to a prescaler stage and a repetition counter. The clock signal fed into the time-base unit passes first through a prescaling stage before reaching the time-base counter.
- Depending on the content of the TIMx_PSC timer prescaler register, the counting signal frequency may be scaled down before reaching the counter stage. The output signal of the prescaling stage is the clock counting signal for the timer counter stage.
- The timer counter is controlled by two timer registers: • The TIMx_CNT timer register is used to read and write the content of the timer counter. • The TIMx_ARR timer register contains the reload value of the timer counter. – If the timer counter is up-counting and it reaches the content of the timer autoreload register (TIMx_ARR), then the timer counter resets itself and a new counting cycle is restarted.
- Each time a new counting cycle is restarted, a timer "update event" is triggered as long as the content of the repetition counter is null. If the content of the repetition counter is not null, then no "update event" is triggered, but a new counting cycle is restarted and the content of the repetition counter is decreased by one. Next to each "update event" the content of the repetition counter is set to the value stored by the TIMx_RCR timer register

The master/slave controller unit

- The master/slave unit provides the time-base unit with the counting clock signal (for example the CK_PSC signal), as well as the counting direction control signal. This unit mainly provides the control signals for the time-base unit.

The timer-channels unit

- The timer channels are the working elements of the timer; they are the means by which a timer peripheral interacts with its external environment
- In general, the timer channels are mapped to the STM32 microcontroller pins
- A timer channel mapped to an STM32 microcontroller pin can be used either as an input or as an output.

The timer-channels unit

- Timer channel configured as output
- When configured as an output, the timer channel is used to generate a set of possible waveforms. As long as the channel is configured in output mode, the content of the TIMx_CCRy channel register is compared to the content of the timer counter.
- The output signal of the channel output stage is mapped to the microcontroller pins as alternate function.

- Timer channel configured as an input
  - When configured as an input, the timer channel can be used to time-stamp the rising and/or the falling edge of external signals. To handle this function, the channel input is mapped to one of the microcontroller pins.

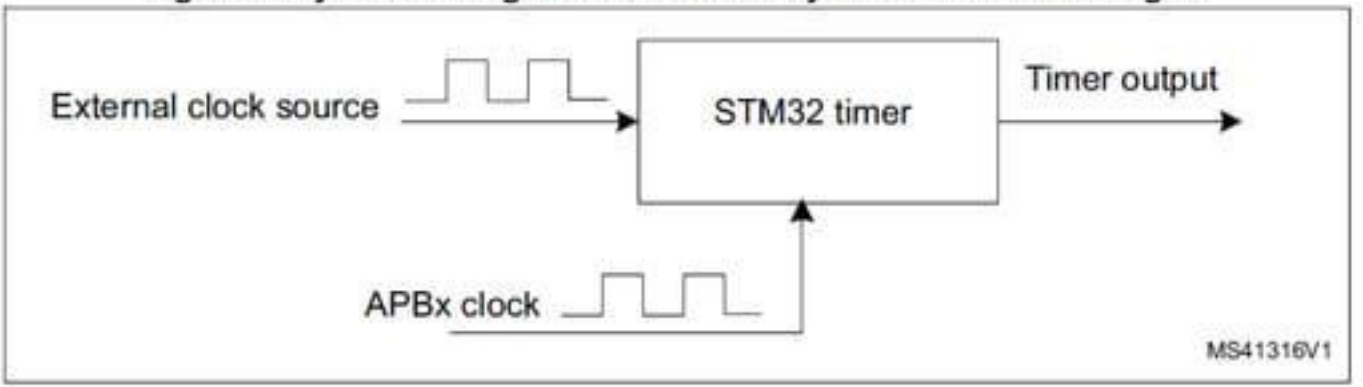
Master slave

- It is possible to configure one slave timer to increment its counter based on a master-timer events such as the timer update event. In this example the master-timer event is signaled by the master timer master/slave controller unit. This controlling unit uses the master timer output-TRGO signal. The master timer output-TRGO signal is connected to the slave timer TRGI-input signal. The master/slave controller unit of the slave timer is configured to use the TRGI-input signal as clock source to increment the slave timer counter.

Timer clocking using external clock-source

- The STM32 timer peripherals can be clocked by an external source clock, but it does not mean that the APB (advanced peripheral bus) clock is not required. An STM32 timer peripheral synchronizes the external clock signal with its own core clock (which is the APB clock). The resulting synchronized clock signal is fed into the timer's prescaler which by turn feeds the timer's counter.

Figure 7. Synchronizing an STM32 timer by an external clock-signal



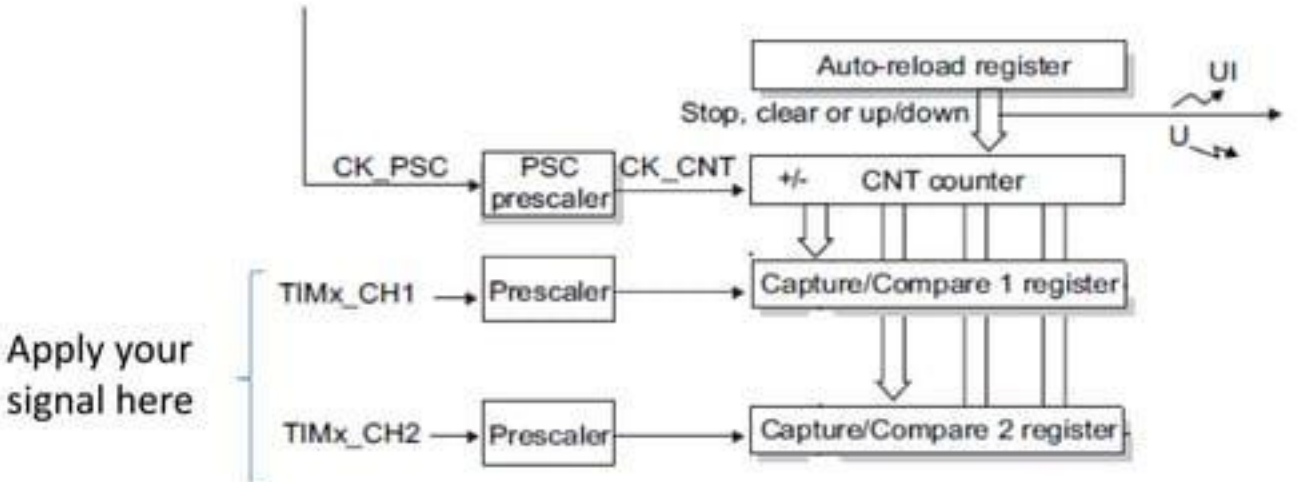
There are two ways to synchronize (or externally clock) an STM32 timer:
• External source clock mode 1: by feeding the external clock signal into one of the timer channel inputs, TIx inputs. • External source clock mode 2: by feeding the external clock signal into the timer ETR input (if it is implemented and available).

Exercise

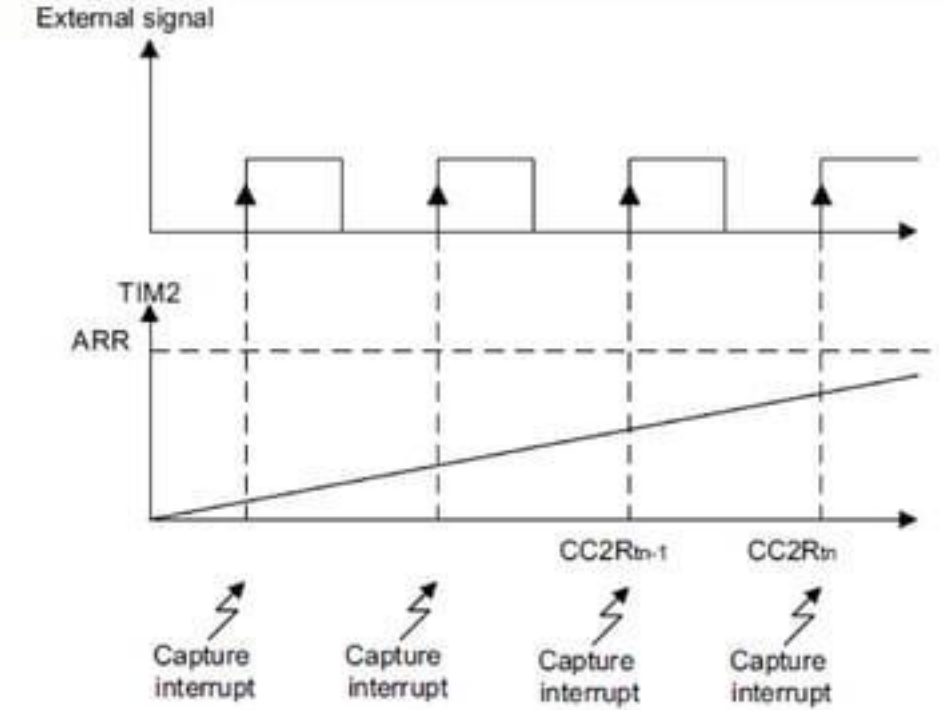
To measure the time period of a signal using a Timer

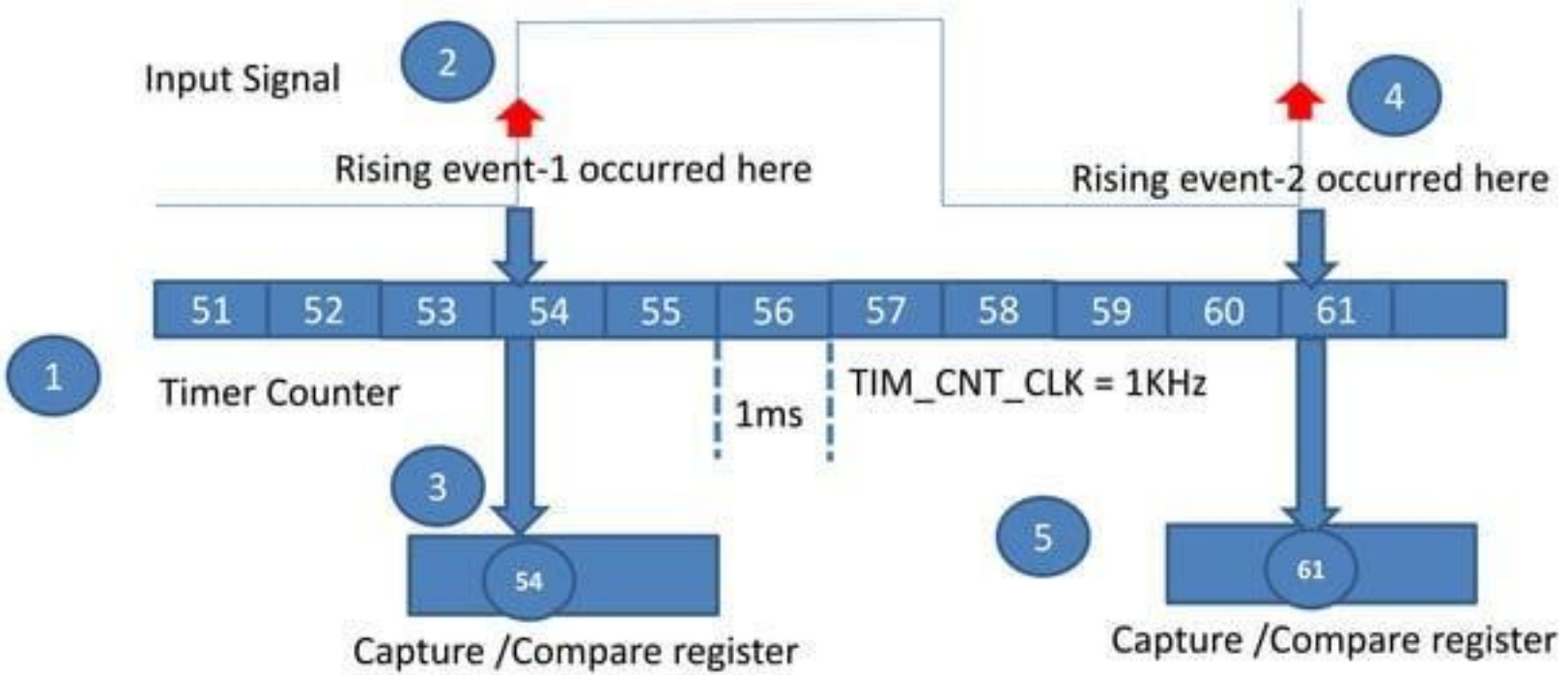
Use LSE and HSI as input signal to Timer

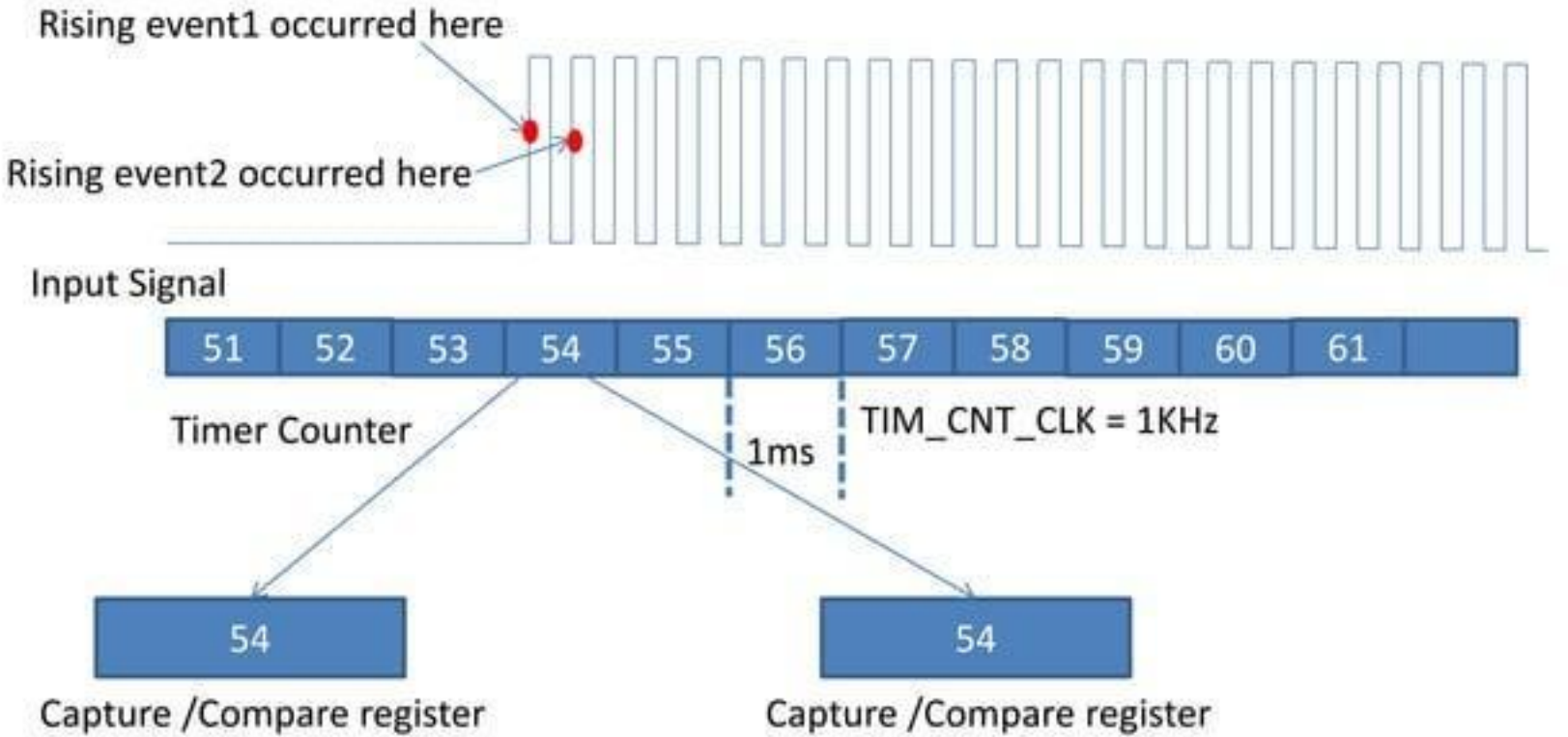
Timer with input capture block

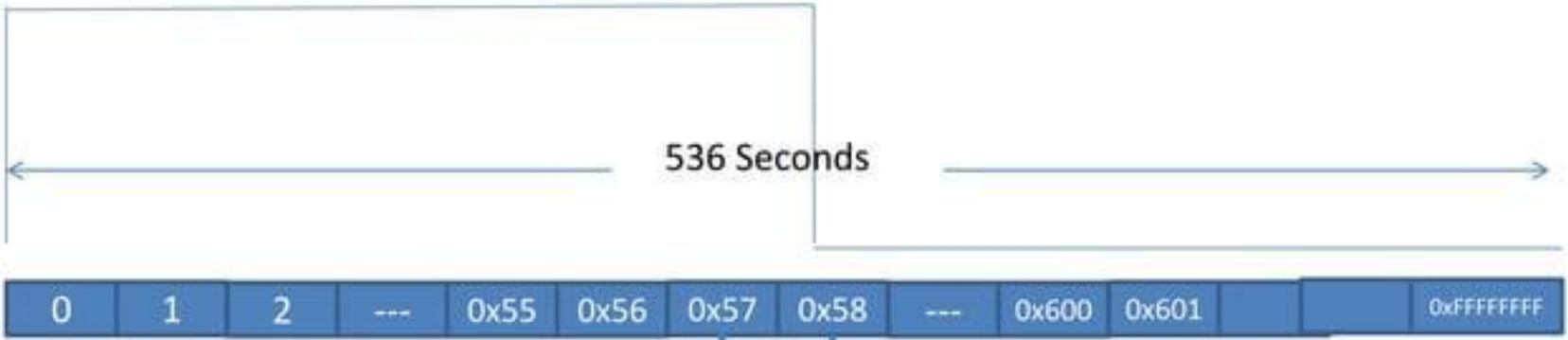


Timing diagram of input capture









32 bit counter

Counts from 0 to 0xFFFFFFFF

Max period = (0xFFFFFFFF+1) * 0.125µS

= 536 Seconds

Min Frequency = 1 /536 = 0.002Hz

Working with Timer Input Channel and STM32 Cube

1) Initialize the TIMER Input Capture Time base

HAL_TIM_IC_Init(TIM_HandleTypeDef *htim)

2) Configure Input Channel of the Timer

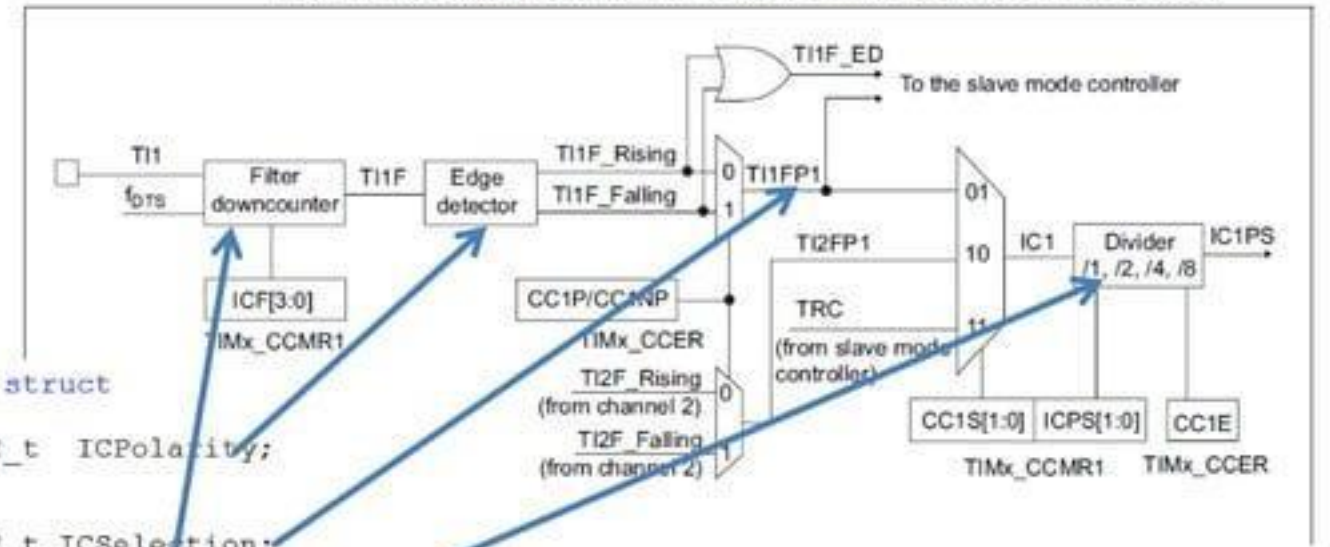
HAL_TIM_IC_ConfigChannel(TIM_HandleTypeDef *htim, TIM_IC_InitTypeDef* sConfig, uint32_t Channel)

STM32 Cube structure for Configuring TIM Input Channel

```
/**
  * @brief  TIM Input Capture Configuration Structure definition
  */

typedef struct
{
  uint32_t  ICPolarity;   /*!< Specifies the active edge of the input signal.
                               This parameter can be a value of @ref TIM_Input_Capture_Polarity */

  uint32_t ICSelection;  /*!< Specifies the input.
                              This parameter can be a value of @ref TIM_Input_Capture_Selection */

  uint32_t ICPrescaler;  /*!< Specifies the Input Capture Prescaler.
                              This parameter can be a value of @ref TIM_Input_Capture_Prescaler */

  uint32_t ICFilter;     /*!< Specifies the input capture filter.
                              This parameter can be a number between Min_Data = 0x0 and Max_Data = 0xF */
} TIM_IC_InitTypeDef;
```

HAL_TIM_IC_ConfigChannel(TIM_HandleTypeDef *htim, TIM_IC_InitTypeDef* sConfig, uint32_t Channel)

Figure 182. Capture/compare channel (example: channel 1 input stage)




```
typedef struct
{
  uint32_t  ICPolarity;

  uint32_t ICSelection;

  uint32_t ICPrescaler;

  uint32_t ICFilter;
} TIM_IC_InitTypeDef;
```

How to measure the period of an input signal ??

- The elapsed time between two consecutive rising edges CC2Rtn - CC2Rtn-1 represents an entire period of the reference signal

- When the capture event occurs, the CC2IF (register TIM2_SR) is set to 1. If the DMA function is enabled, it will generate a DMA request. If the capture occurs, CC2IF flag has been set, then the over sampling flag CC2OF is set.
- When a rising edge comes, numerical timer current meter (TIM2_CNT) will write in the TIM2_CCR2. Wait until the next rising edge to, there will be another counter value in TIM2_CNT records. According to the two data value, we can calculate the cycle of input data. Overflow timer is not allowed.

Working with Timer Output Channels and STM32 Cube

1) Initialize the TIMER output Compare Time base

HAL_TIM_OC_Init(TIM_HandleTypeDef *htim)

2) Configure output channel of the timer

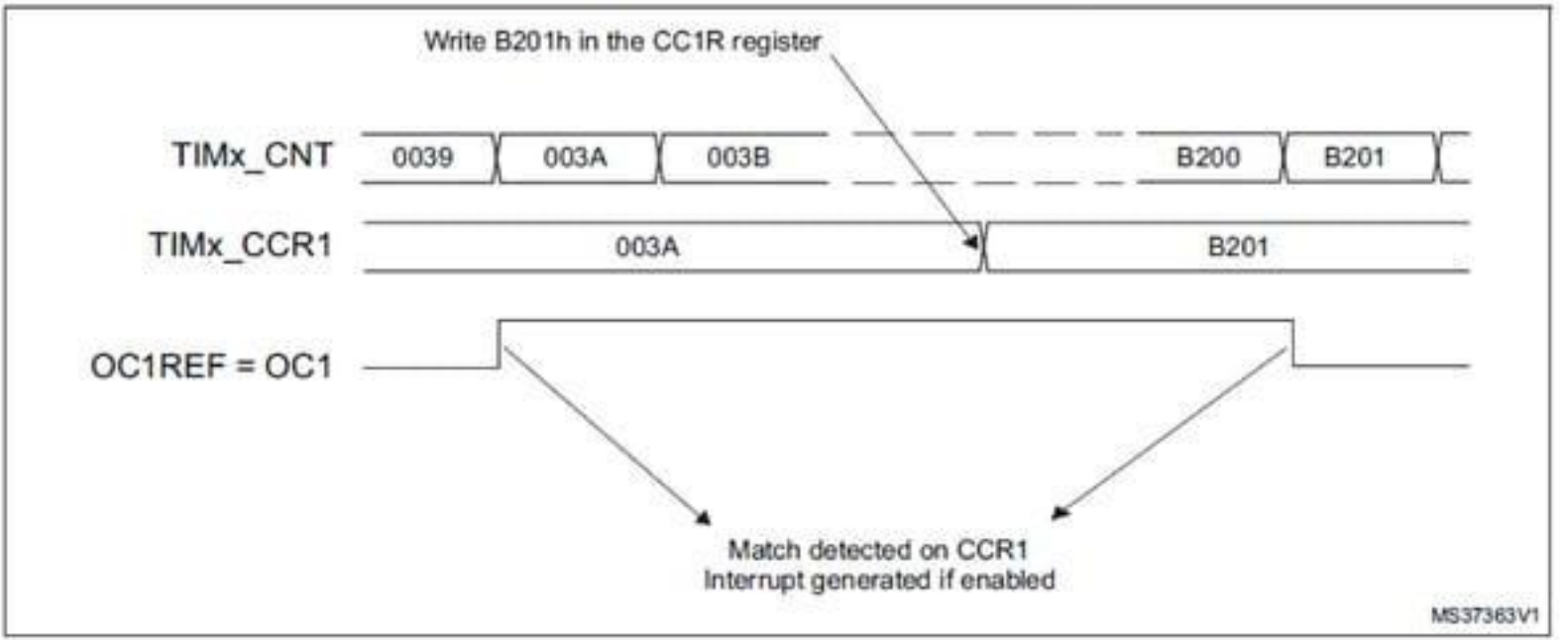
HAL_TIM_OC_ConfigChannel(TIM_HandleTypeDef *htim, TIM_OC_InitTypeDef* sConfig, uint32_t Channel)

STM32 Cube structure for Configuring TIM Input Channel

```
/**
  * @brief  TIM Input Capture Configuration Structure definition
  */

typedef struct
{
  uint32_t  ICPolarity;   /*!< Specifies the active edge of the input signal.
                               This parameter can be a value of @ref TIM_Input_Capture_Polarity */

  uint32_t ICSelection;  /*!< Specifies the input.
                              This parameter can be a value of @ref TIM_Input_Capture_Selection */

  uint32_t ICPrescaler;  /*!< Specifies the Input Capture Prescaler.
                              This parameter can be a value of @ref TIM_Input_Capture_Prescaler */

  uint32_t ICFilter;     /*!< Specifies the input capture filter.
                              This parameter can be a number between Min_Data = 0x0 and Max_Data = 0xF */
} TIM_IC_InitTypeDef;
```

HAL_TIM_IC_ConfigChannel(TIM_HandleTypeDef *htim, TIM_IC_InitTypeDef* sConfig, uint32_t Channel)

Figure 186. Output compare mode, toggle on OC1



Timer Modes

1. Basic Time base generation
2. Input Capture Mode
3. Output Compare Mode
4. PWM Mode
5. One Pulse Mode

TIMER Output Compare Mode

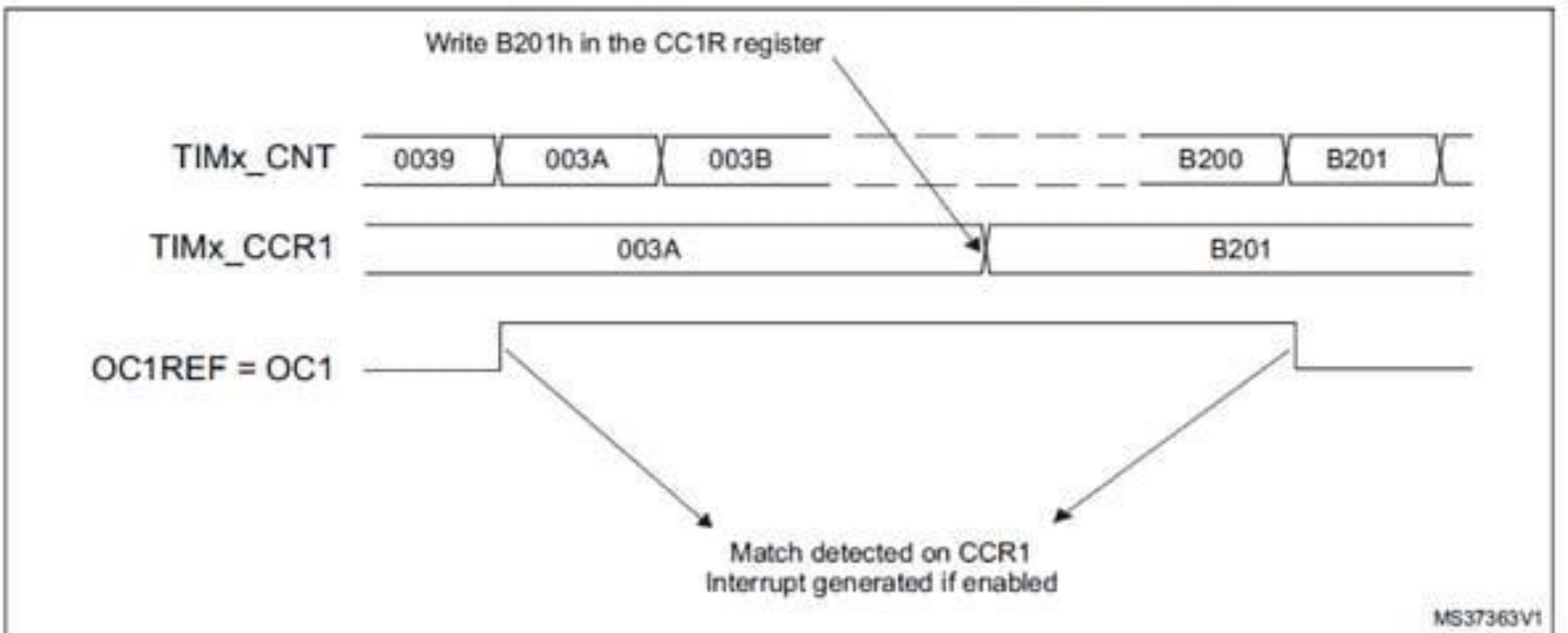
Exercise :

Write an application by using TIMER2 , to produce square wave forms of

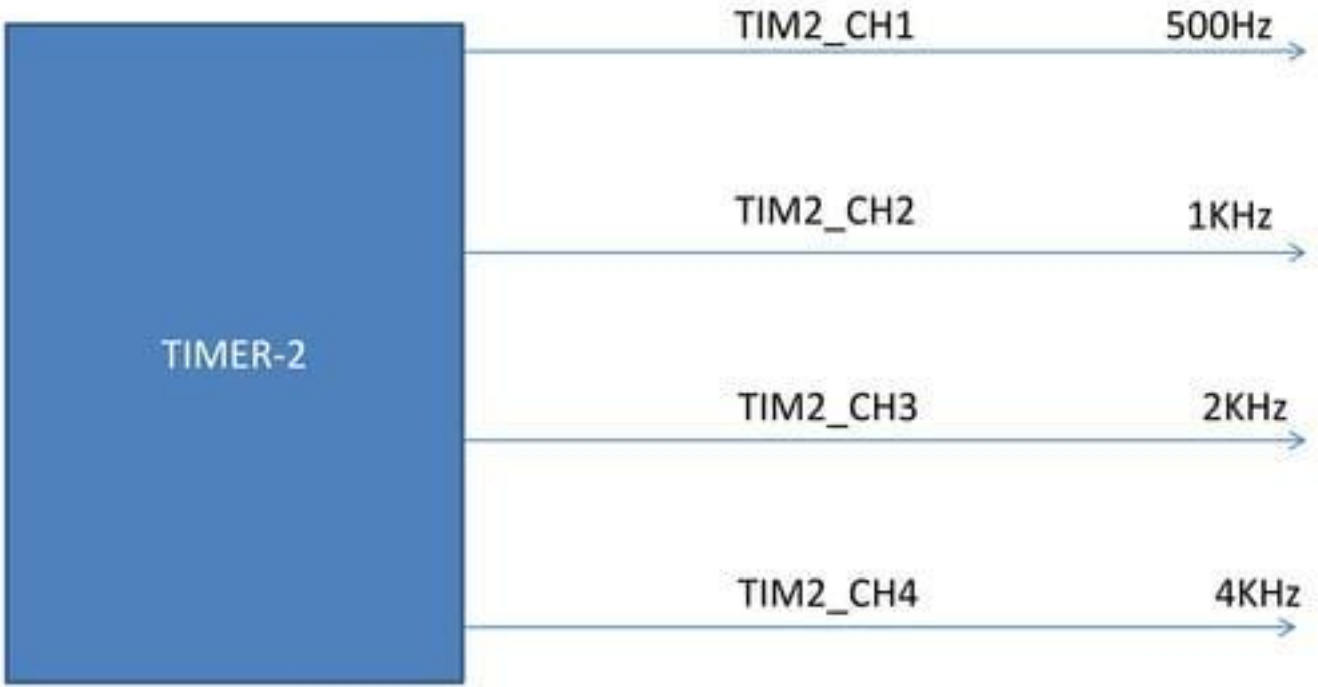
500Hz, 1KHz, 2KHz and 4KHz on 4 different output channels.

TIM_OCMODE_TOGGLE

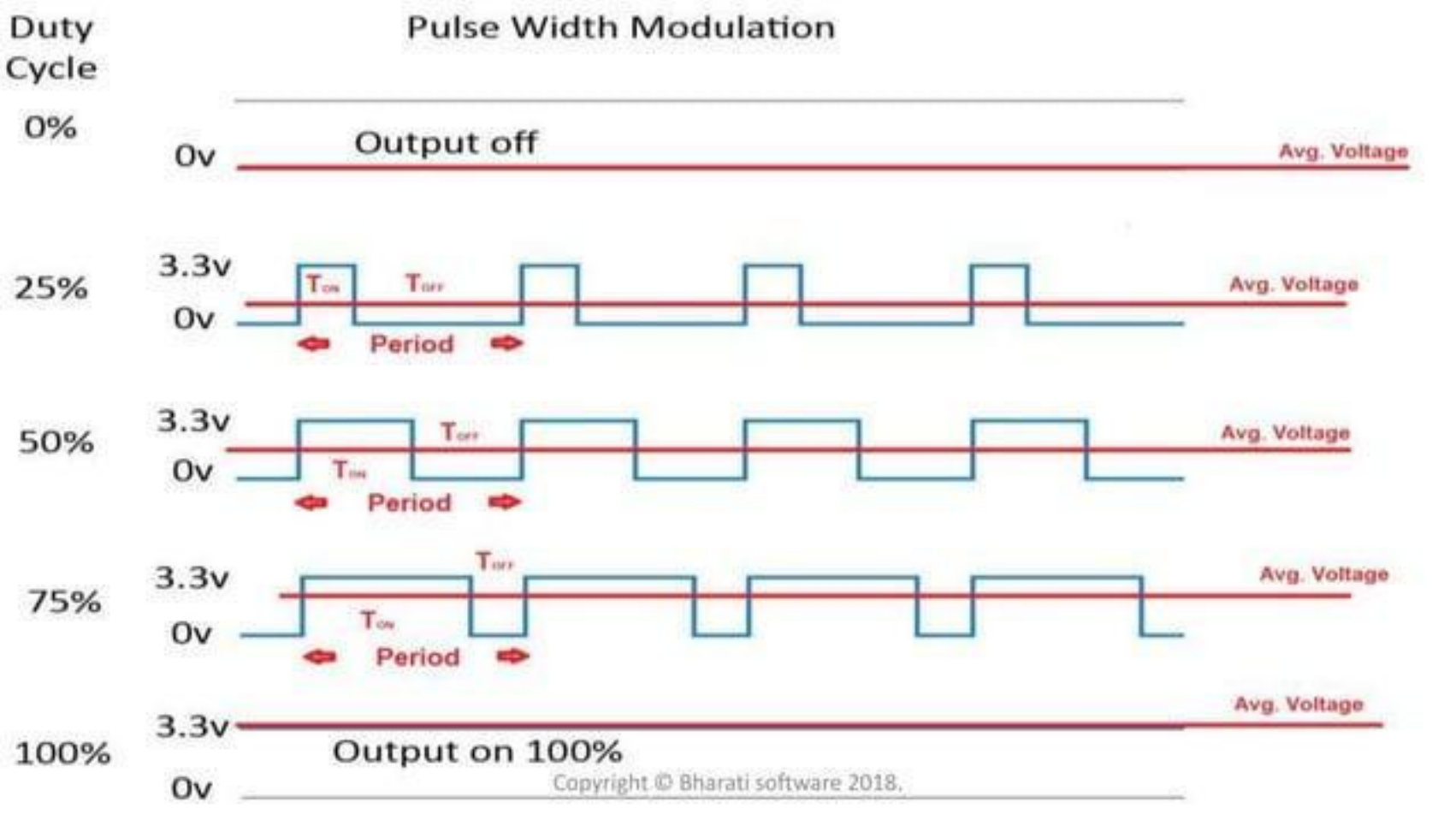
Figure 186. Output compare mode, toggle on OC1



TIMER Output Compare Mode



How are we going to produce waveforms using OC mode?


Duty Cycle
Pulse Width Modulation
0%
0v
Output off
Avg. Voltage
25%
3.3v
0v
Ton
Toff
Period
Avg. Voltage
50%
3.3v
0v
Toff
Ton
Period
Avg. Voltage
75%
3.3v
0v
Toff
Ton
Period
Avg. Voltage
100%
3.3v
0v
Output on 100%
Avg. Voltage
Copyright © Bharati software 2018.

Exercise

By using PWM mode, generate PWM signals of 25%, 45%, 75% and 90% Duty cycle on TIMER2's channels .

Controller Area Network

- CAN Bus Protocol and its features (Fundamentals)
- Operating Principles of the CAN Bus
- Different Message Formats
- Working with STM32 bxCAN Peripherals
- Various Examples

Intro to CAN Bus

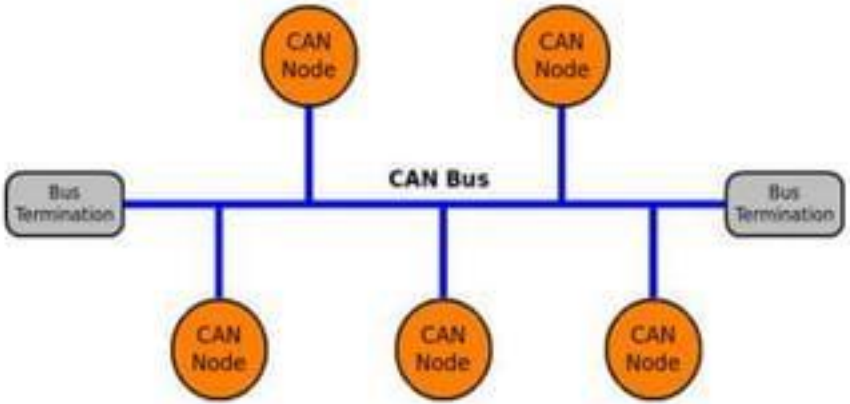
- The Controller Area Network Protocol was originally developed during the late 1980's by the German company Robert Bosch for the automotive industry
- It is a multi-master serial communication bus whose basic design specification called for high speed, high noise-immunity and error-detection features
- CAN offers data communication up to 1 Mbit/sec
- the Error Confinement and the Error Detection features make it more reliable in noise-critical environments. In the automotive industries

CAN's most attractive features

- Low cost
- Extreme robustness
- High data transmission speeds (up to 1 MBit/sec).
- Reliability. Excellent error handling and Error Confinement abilities.
- Automatic re-transmission of faulty messages.
- Automatic bus disconnection of nodes that are suspected to be physically faulty.
- Functional addressing – data messages do not contain source or destination addresses, only identifiers relating to their function and/or priority.

CAN's most attractive features

- **It's a broadcast type of Bus(**Unlike a traditional network such as USB or Ethernet, or i2c , CAN does not send data point-to-point from node A to node B under the supervision of a central bus master)



It's a broadcast type of Bus

All devices can hear the transmission

No way to send a data specifically to a no
its address or something

All nodes will pick up the traffic on the bus

The CAN standard defines a communication network that links all the nodes connected to a bus and enables them to talk with one another. There may or may not be a central control node, and nodes may be added at any time, even while the network is operating (hot-plugging).

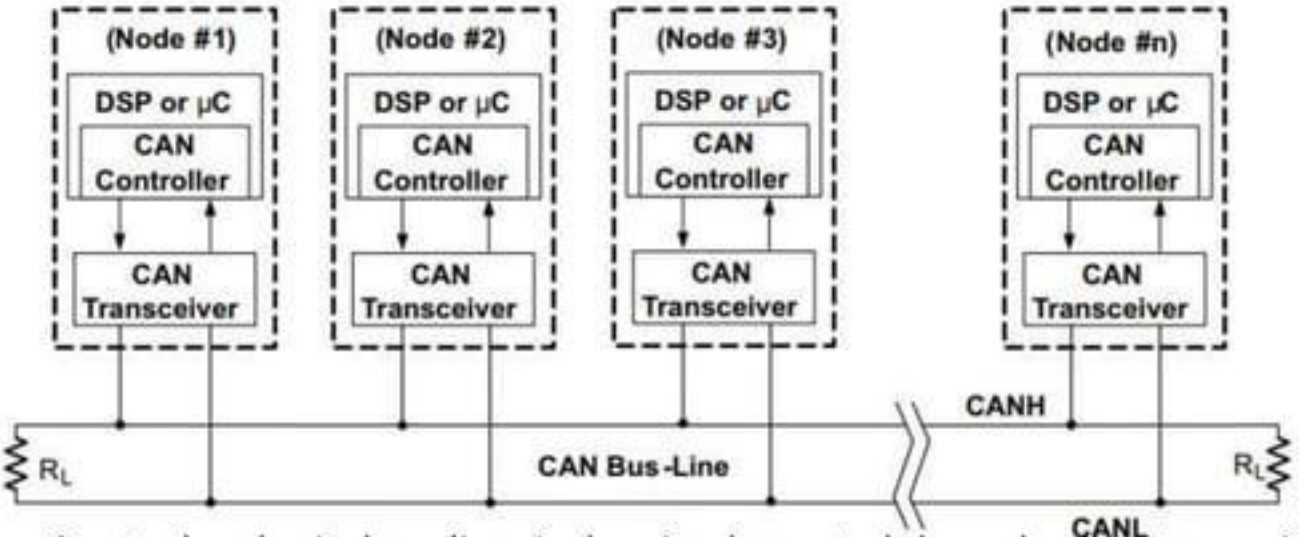
Closer look at the CAN node

Note that a transmitting node constantly monitors each bit of its own transmission

This is the reason for the transceiver configuration of Figure 4 in which the CANH and CANL output pins of the driver are internally tied to the receiver's input

Because each node continuously monitors its own transmissions,

Details of a CAN Bus



r. Connection to the physical medium is then implemented through a line transceiver such as TI's SN65HVD230 3.3-V CAN transceiver to form a system node as shown in

CAN signaling

- Signalling is differential which is where CAN derives its robust noise immunity and fault tolerance
- Balanced differential signalling reduces noise coupling and allows for high signalling rates over twisted-pair cable
- Balanced means that the current flowing in each signal line is equal but opposite in direction, resulting in a field-cancelling effect that is a key to low noise emissions
- The use of balanced differential receivers and twisted-pair cabling enhance the common-mode rejection and high noise immunity of a CAN bus.
- The cable is specified to be a shielded or unshielded twisted-pair with a 120-Ω characteristic impedance (Zo)
- The ISO 11898 Standard defines a single line of twisted-pair cable as the network topology as shown in, , terminated at both ends with 120-Ω resistors. which match the characteristic impedance of the line to prevent signal reflections

CAN signaling

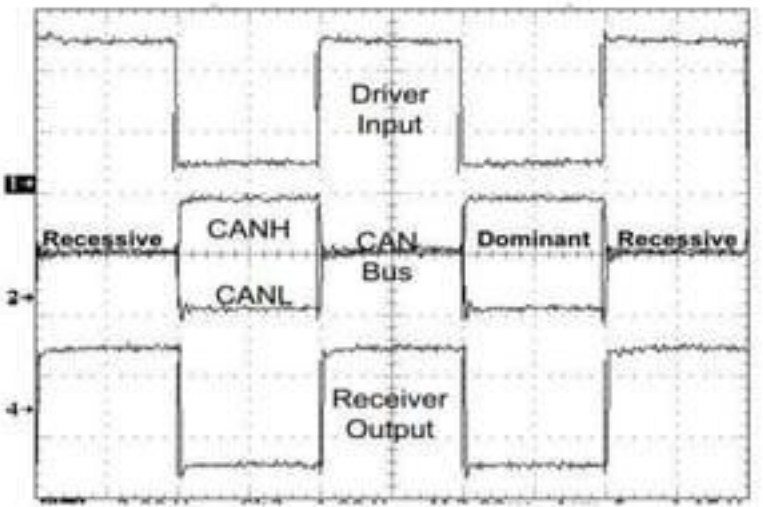


Figure 7. CAN Dominant and Recessive Bus States

CAN signaling

A CAN bus is based in differential signalling. The two lines, CAN-High (CAN+) and CAN-Low (CAN-), are both at the same potential when the bus is idle. To send bits, a CAN transmitter puts a differential voltage on the lines of about 2 volts.

A CAN transmitter first sees if the bus is idle and if it is, starts to transmit. How the arbitration works is that a transmitter monitors the bus as it's transmitting. Transmission is done as above by either keeping the two lines at the same potential or at a differential potential. So if the transmitter transmits a bit by keeping the lines at the same potential (sic), but it sees that the two transmit lines have a differential potential, that means that some other node is transmitting and the first transmitter has lost the arbitration. It must then stop transmitting.

When a node first starts transmitting, the bits transmitted are the same until the address of the transmitting node which are obviously different. If two nodes start transmitting together, they will transmit together in sync till the address part is

The Bit Fields of Standard CAN and Extended CAN

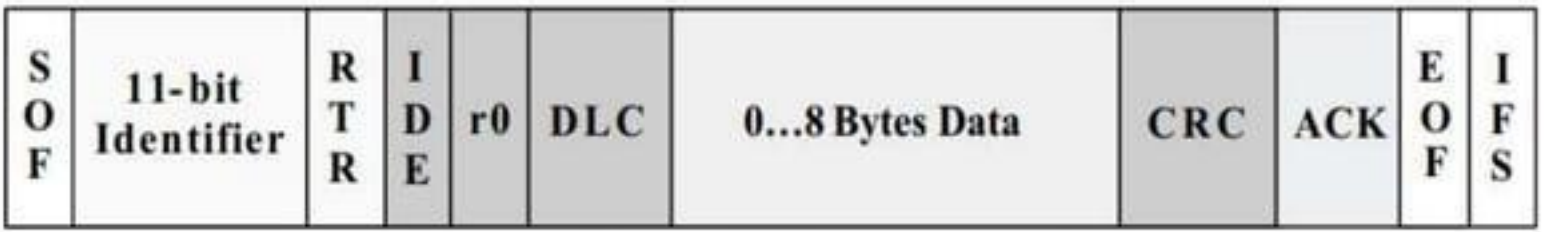


Figure 2. Standard CAN: 11-Bit Identifier

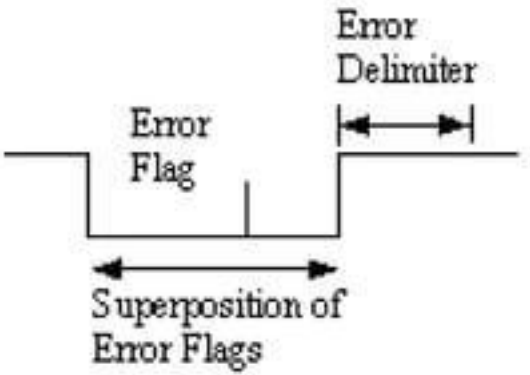
The Error Frame

- The error frame is a special message that violates the formatting rules of a CAN message
- It is transmitted when a node detects an error in a message, and causes all other nodes in the network to send an error frame as wel
- The original transmitter then automatically retransmits the message
- An elaborate system of error counters in the CAN controller ensures that a node cannot tie up a bus by repeatedly transmitting error frames.
- It is transmitted when a node detects a fault and will cause all other nodes to detect a fault – so they will send Error Frames, too

The Error Frame

- The Error Frame consists of an Error Flag, which is 6 bits of the same value (thus violating the bit-stuffing rule) and an Error Delimiter, which is 8 recessive bits
- The Error Delimiter provides some space in which the other nodes on the bus can send their Error Flags when they detect the first Error Flag
- an active error frame consists of six dominant bits—violating the bit stuffing rule
- This is interpreted as an error by all of the CAN nodes which then generate their own error frame.
- It is important to note that the retransmitted message still has to contend for arbitration on the bus

Error Frame



A valid frame

- A message is considered to be error free when the last bit of the ending EOF field of a message is received in the error-free recessive state. A dominant bit in the EOF field causes the transmitter to repeat a transmission

error-checking procedure

- The CAN protocol incorporates five methods of error checking:
  - three at the message level and two at the bit level
- If a message fails any one of these error detection methods, it is not accepted and an error frame is generated from the receiving node
- This forces the transmitting node to resend the message until it is received correctly
- y. However, if a faulty node hangs up a bus by continuously repeating an error, its transmit capability is removed by its controller after an error limit is reached

error-checking procedure

- Error checking at the message level is enforced by the CRC and the ACK slots displayed in Figure 2 and Figure 3.
- The 16-bit CRC contains the checksum of the preceding application data for error detection with a 15-bit checksum and 1-bit delimiter.
- . The ACK field is two bits long and consists of the acknowledge bit and an acknowledge delimiter bit.
- Also at the message level is a form check. This check looks for fields in the message which must always be recessive bits. If a dominant bit is detected, an error is generated. The bits checked are the SOF, EOF, ACK delimiter, and the CRC delimiter bits
- At the bit level, each bit transmitted is monitored by the transmitter of the message. If a data bit (not arbitration bit) is written onto the bus and its opposite is read, an error is generated. The only exceptions to this are with the message identifier field which is used for arbitration, and the acknowledge slot which requires a recessive bit to be overwritten by a dominant bit.
- The final method of error detection is with the bit-stuffing rule where after five consecutive bits of the same logic level, if the next bit is not a complement, an error is generated

The Overload Frame

- It is very similar to the Error Frame with regard to the format and it is transmitted by a node that becomes too busy.

Arbitration

- Bus access is event-driven and takes place randomly. If two nodes try to occupy the bus simultaneously, access is implemented with a nondestructive, bit-wise arbitration
- Nondestructive means that the node winning arbitration just continues on with the message, without the message being destroyed or corrupted by another node.
- the Arbitration Field, which determines the priority of the message when two or more nodes are contending for the bus.
- \The Arbitration Field contains:For CAN 2.0A, an 11-bit Identifier and one bit, the RTR bit, which is dominant for data frames.
- For CAN 2.0B, a 29-bit Identifier (which also contains two recessive bits: SRR and IDE) and the RTR bit.

Arbitration

- . An actual application may include a temperature sensor sending out a temperature update that is used to adjust the motor speed of a fan. If a pressure sensor node wants to send a message at the same time, arbitration ensures that the message is sent.
- sending a '0' is dominant over a '1', which results in the lowest address winning arbitration

Arbitration

- Example : Will be most grateful if you could just clear this up: Lets say the identifiers are "1011001" and "1000110" when the third bit is reached the first transmitter is sending "1" and second one is sending "0"; so as per CAN protocol the dominant bit is 0, and it overwrites the recessive bit. So now the Bus holds "0"; the first module detects this and will stop transmission while the second one will keep on transmitting. Is my understanding correct ??

Message Transfer : Example

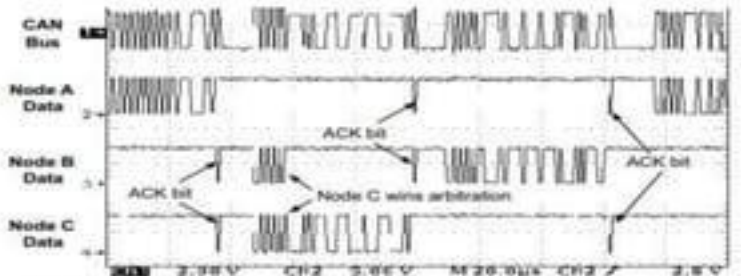


Figure 6. CAN Bus Traffic

Message priority

- The allocation of priority to messages in the identifier is a feature of CAN that makes it particularly attractive for use within a real-time control environment
- The lower the binary message identifier number, the higher its priority
- . An identifier consisting entirely of zeros is the highest priority message on a network because it holds the bus dominant the longest
- Therefore, if two nodes begin to transmit simultaneously, the node that sends a last identifier bit as a zero (dominant) while the other nodes send a one (recessive) retains control of the CAN bus and goes on to complete its message. A dominant bit always overwrites a recessive bit on a CAN bus.
- The allocation of message priority is up to a system designe

CAN arbitration process

- handled automatically by a CAN controller
- Because each node continuously monitors its own transmissions,

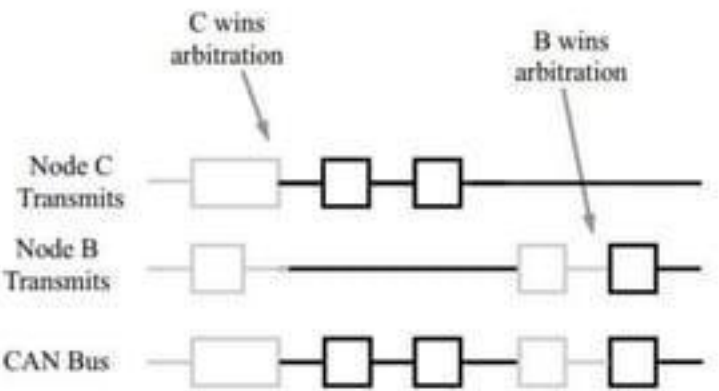
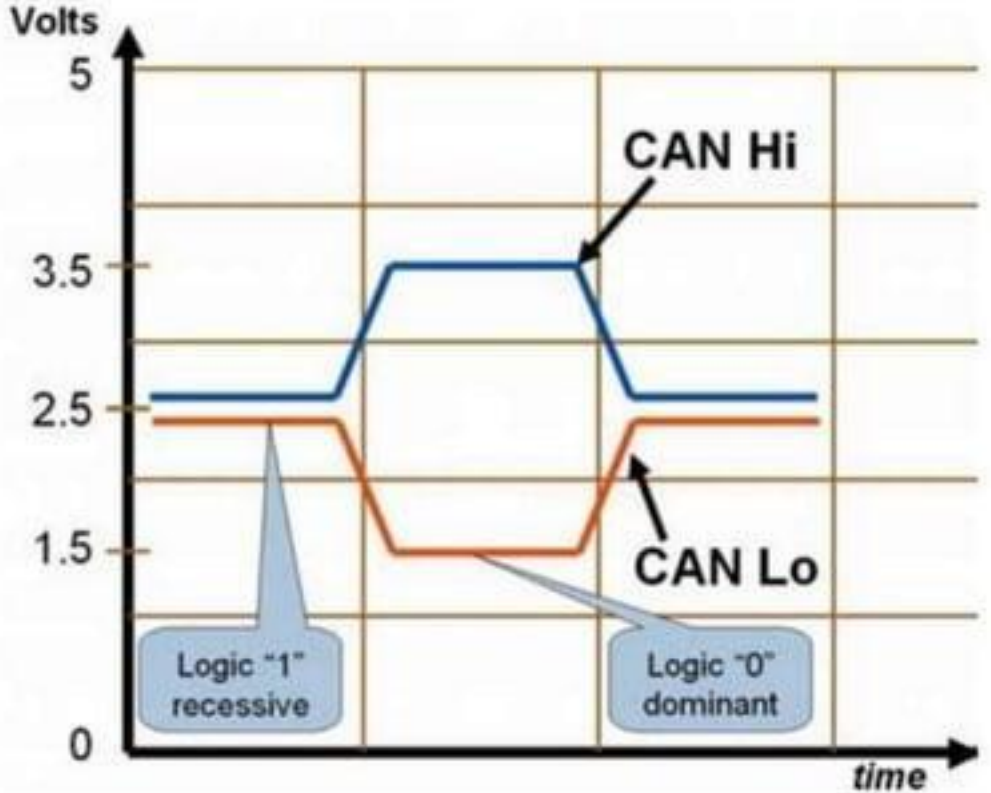
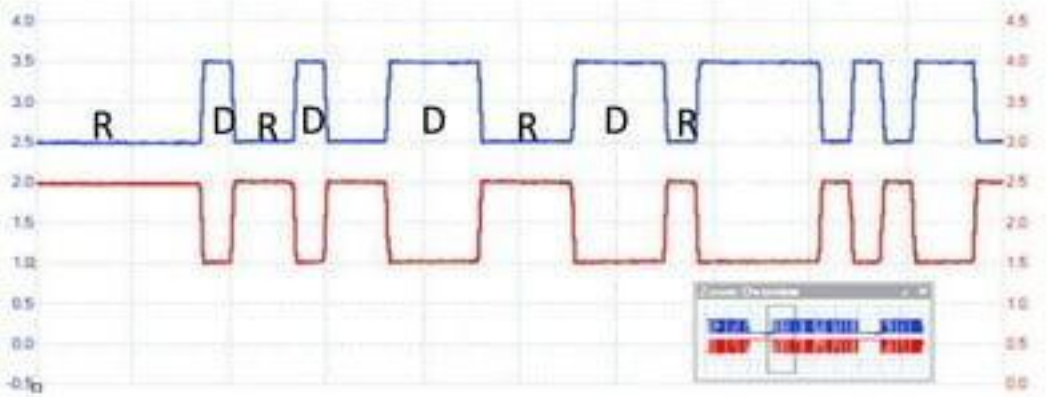


Figure 5. Arbitration on a CAN Bus

CAN arbitration process

- displays the CAN arbitration process that is handled automatically by a CAN controller. Because each node continuously monitors its own transmissions, as node B's recessive bit is overwritten by node C's higher priority dominant bit, B detects that the bus state does not match the bit that it transmitted. Therefore, node B halts transmission while node C continues on with its message. Another attempt to transmit the message is made by node B once the bus is released by node C. This functionality is part of the ISO 11898 physical signaling layer, which means that it is contained entirely within the CAN controller and is completely transparent to a CAN user.





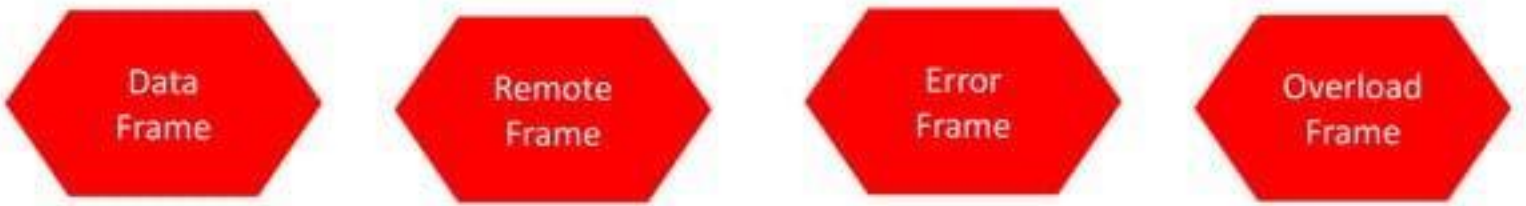
Node A

Node B		dominant	recessive
	dominant	dominant	dominant
	recessive	dominant	recessive

	dominant	recessive
dominant	dominant	dominant
recessive	dominant	recessive

CAN Message Types

- There are 4 different message types(or frames) in CAN Protocol



Data Frame

Most Common Message Types in CAN Communication .

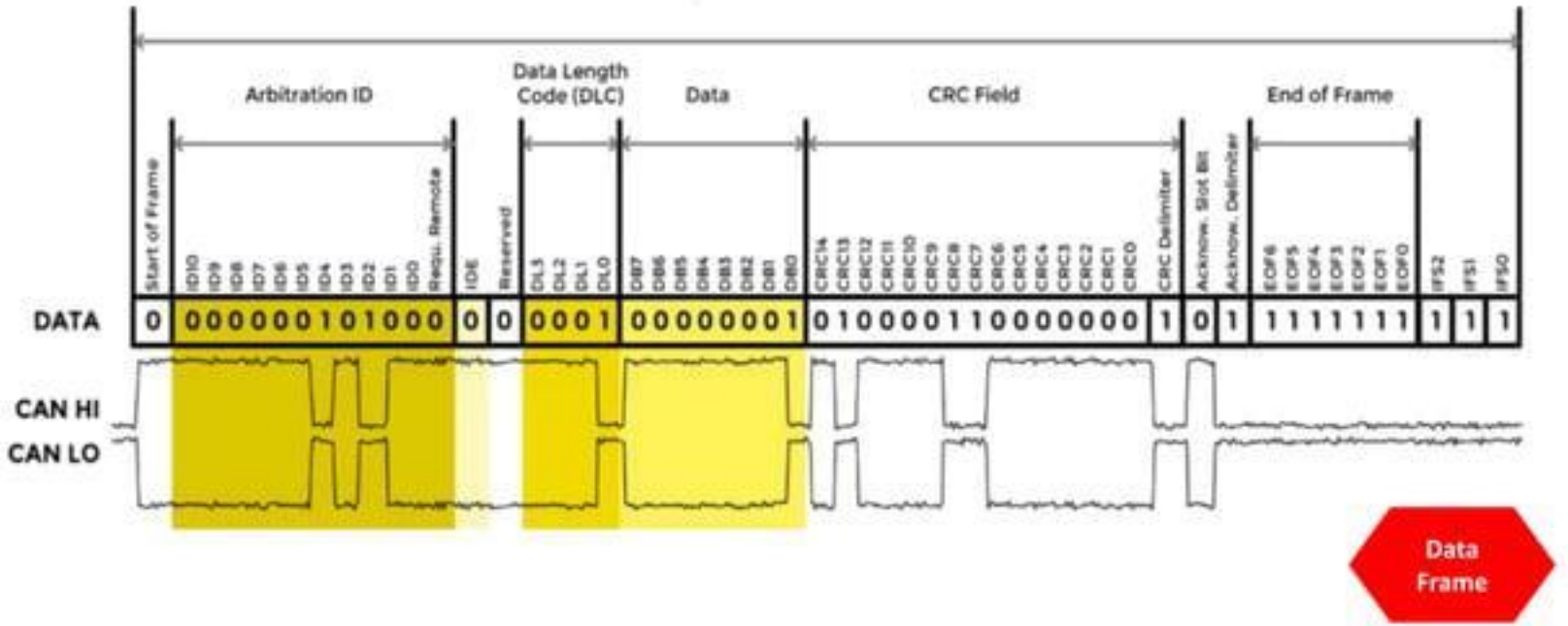
We use it very frequently

A Node uses this frame to send a message to other nodes on the Can bus.

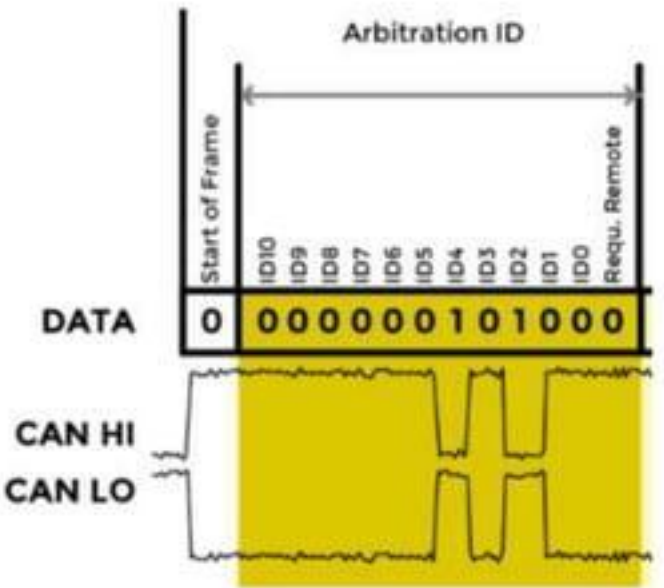
CAN Node

Here is a message !

Data Frame Format



Arbitration Field



The Arbitration Field, which determines the priority of the message when two or more nodes are contending for the bus. The Arbitration Field contains: For CAN 2.0A, an 11-bit Identifier and one bit, the RTR bit, which is dominant for data frames.

Identifier establishes the priority of the message. The lower the binary value, the higher its priority.

Standard CAN Vs Extended CAN

- The original specification is the Bosch specification Version 2.0
- Version 2.0 specification is divided into two parts
  - **Standard CAN (Version 2.0A). Uses 11 bit identifiers.**
  - **Extended CAN (Version 2.0B). Uses 29 bit identifiers.**
- The difference between these two formats is that the length of bits, i.e., the standard CAN Frame format supports 11-bits length for the identifier, whereas the extended frame supports 29-bits length for the identifier, which is made up of 18-bit extension and an 11-bit identifier.
- Most 2.0A controllers transmit and receive only Standard format messages.
- 2.0B controllers can send and receive messages in both formats.

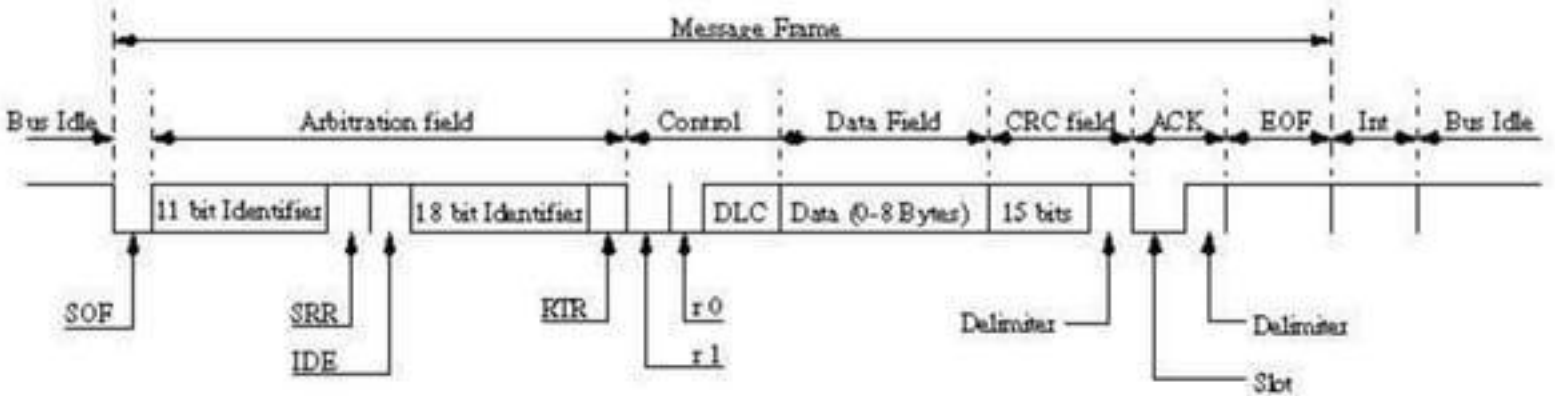
Standard CAN Vs Extended CAN

- If you have a CAN network which Consists of both 2.0A and 2.0B based CAN devices and if you use extended frame format (29 bit identifiers), then your network will not work, because 2.0A devices will generate an error

- 2.0B controllers are completely backward compatible with 2.0A controllers and can transmit and receive messages in either format.

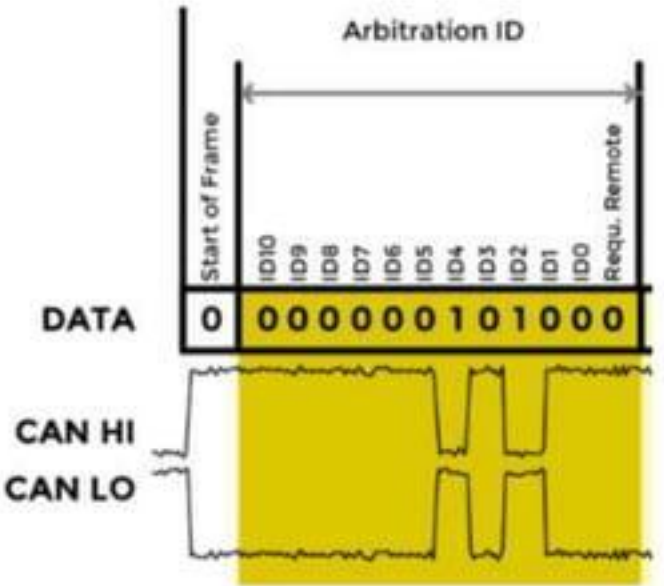
- 2.0A controller based devices are capable of transmitting and receiving only messages in 2.0A format (standard format). With this type of controller, reception of any 2.0B message will flag an error.

Extended Frame Format or CAN 2.0 B



The IDE bit differs CAN extended frame format and the CAN standard frame format wherein IDE is transmitted as dominant in an 11-bit frame case and recessive in a 29-bit frame case.

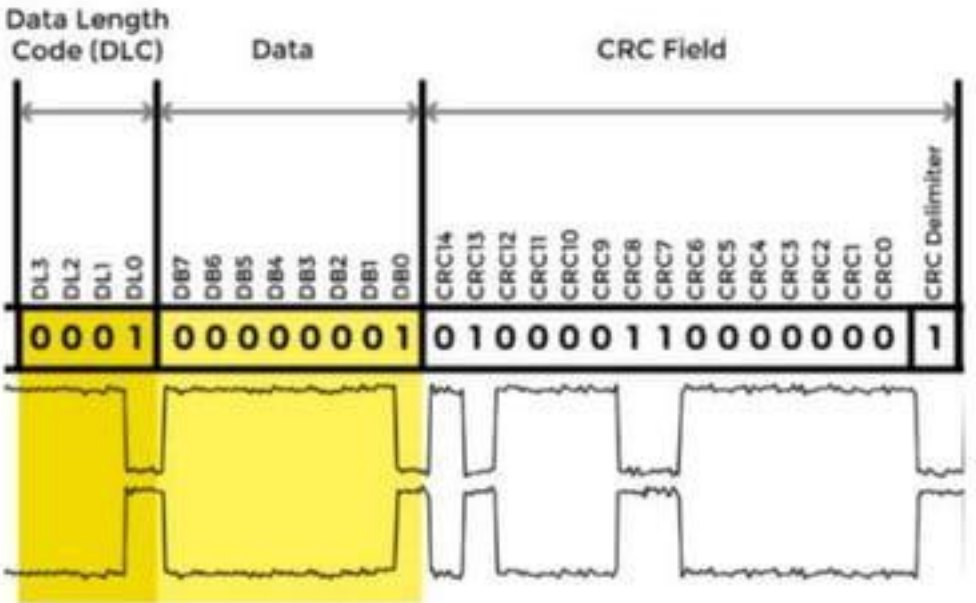
RTR bit (Remote Transmission Request)



A dominant (logic 0), RTR bit indicates that the message is a Data Frame

A recessive (logic 1) value indicates that the message is a Remote Transmission Request (known as Remote Frame.) A Remote Frame is a request by one node for data from some other node on the bus. Remote Frames do not contain a Data Field.

DLC, DATA and CRC Fields

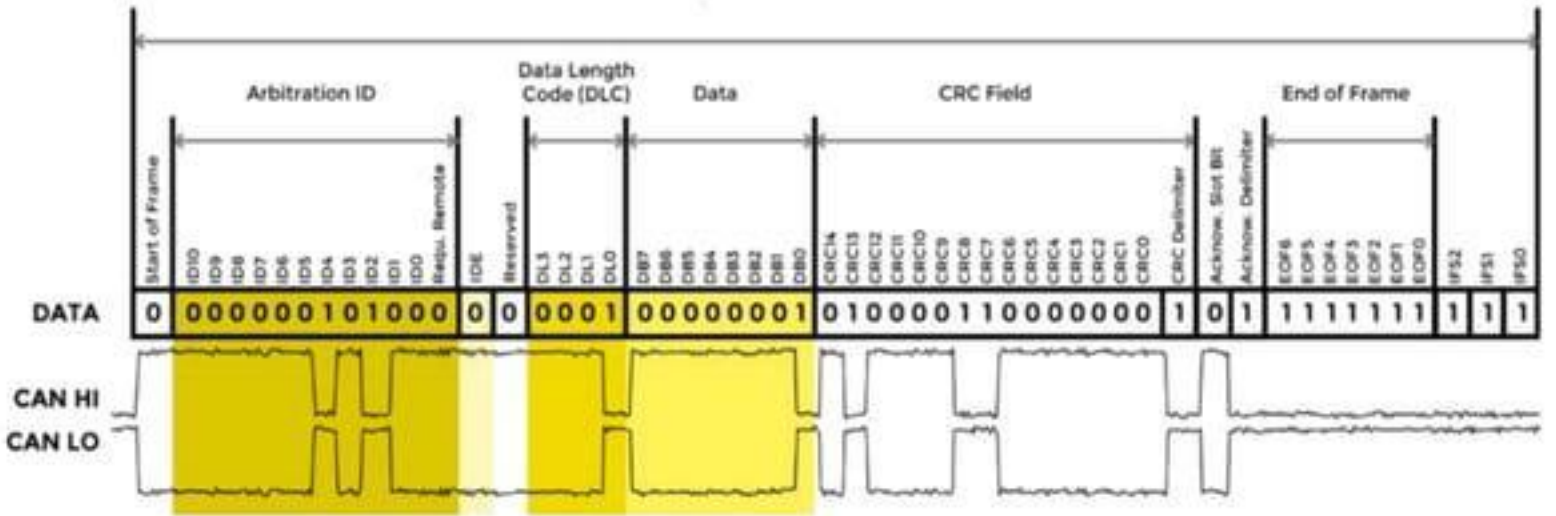


The 4-bit data length code (DLC) contains the number of bytes of data being transmitted

The Data Field, which contains zero to eight bytes of data.

The CRC Field contains a 15-bit checksum calculated on most parts of the message. This checksum is used for error detection

ACK Bit

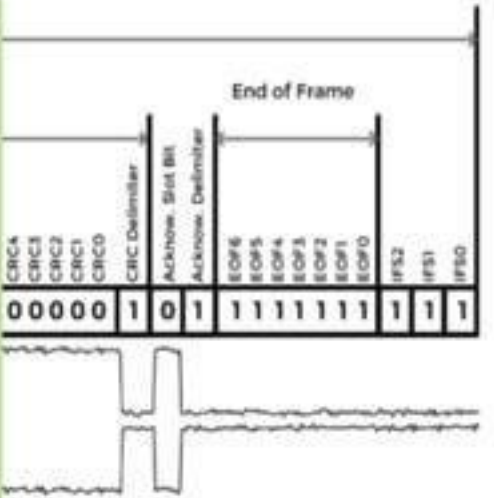


ACK Bit

It is worth noting that the presence of an Acknowledgement Bit on the bus does not mean that any of the intended devices has received the message.

It just means that one or more nodes on the bus has received it correctly and Transmitter concludes that message sent successfully .

If Transmitter sees recessive state at the ACK slot , then it retransmits the message until it sees dominant state. That's the reason when there is only one node on the bus, transmitter keep sending the same message since no one is there to ack it.



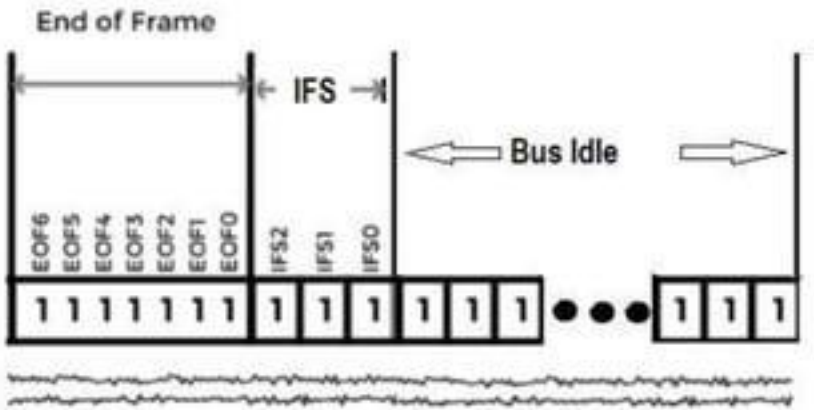
ACK Significance

- Every node receiving an accurate message overwrites this recessive bit in the original message with a dominate bit, indicating an error-free message has been sent. If a receiving node detects an error then it leaves this bit recessive, it discards the message and the sending node repeats the message after re-arbitration. In this way, each node acknowledges (ACK) the integrity of its data. ACK is 2 bits, one is the acknowledgment bit and the second is a delimiter
- Because all receivers must participate in the acknowledgment algorithm regardless of whether the message is intended for them or not, an acknowledgment to the transmitter may occur even if the expected receiver is not present on the network
- This means that the CAN Acknowledgment does not guarantee that a data transfer has occurred between the transmitter and a designated receiver. It does not confirm that a requested action has been understood or performed. CAN Acknowledgment only confirms that all resident network nodes agree that the CAN message did not violate any Data Link Layer rules.

End of Frame and IFS

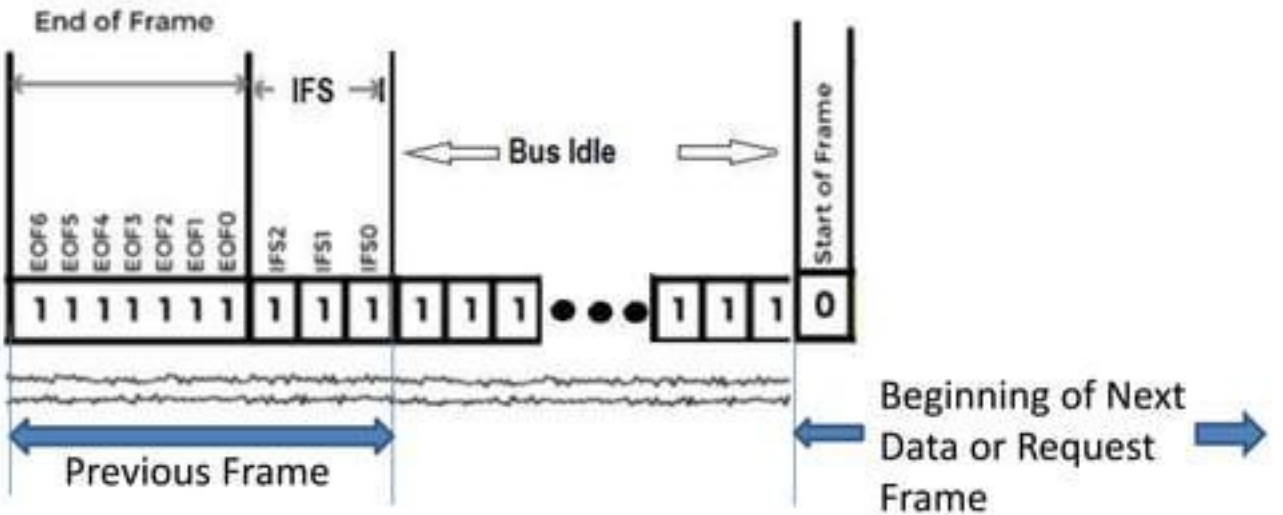
End of Frame : 7 Recessive State

IFS (Inter Frame Spacing) : 3 Recessive state

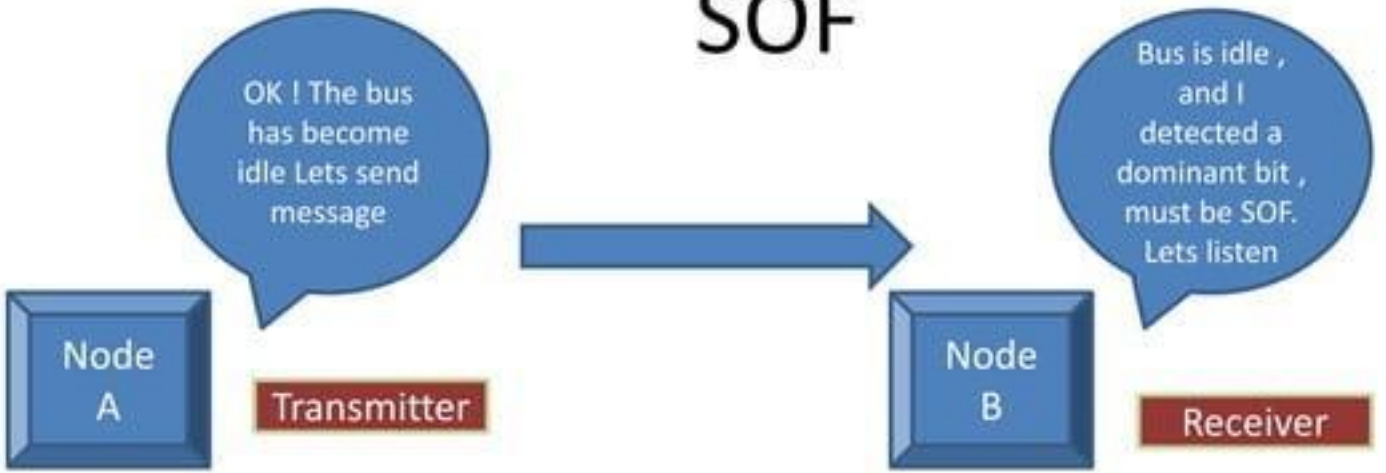


SOF

SOF–The single dominant start of frame (SOF) bit marks the start of a message, and is used to synchronize the nodes on a bus after being idle.



SOF



*sends a message using data frame .
The first bit of the frame will be dominant which marks the start of frame(SOF)

*when Bus is idle, it detects a dominant bit(SOF) , hence understands that someone is transmitting

Remote Frame

- The intended purpose of the remote frame is to solicit the transmission of data from another node
- The remote frame is similar to the data frame, with two important differences. First, this type of message is explicitly marked as a remote frame by a recessive RTR bit in the arbitration field, and secondly, there is no data.

CAN Node

Consider this request and send me some data

Remote Frame

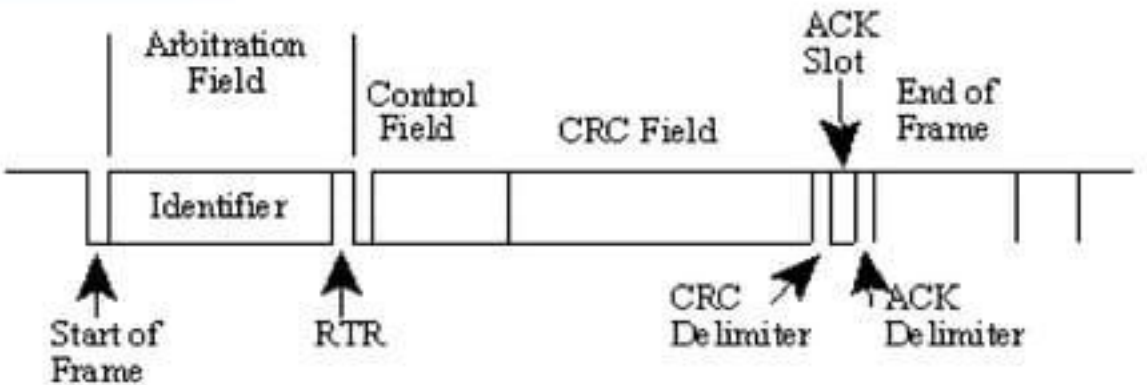
- If, say, node A transmits a Remote Frame with the Arbitration Field set to 123, then node B, if properly initialized, might respond with a Data Frame with the Arbitration Field also set to 123.
- Remote Frames can be used to implement request-response type communication between nodes.
- Most CAN controllers can be programmed either to automatically respond to a Remote Frame

CAN Node

Consider this request and send me some data

Remote Frame

RTR bit is Recessive and there is no Data



BUS access from Multiple Nodes

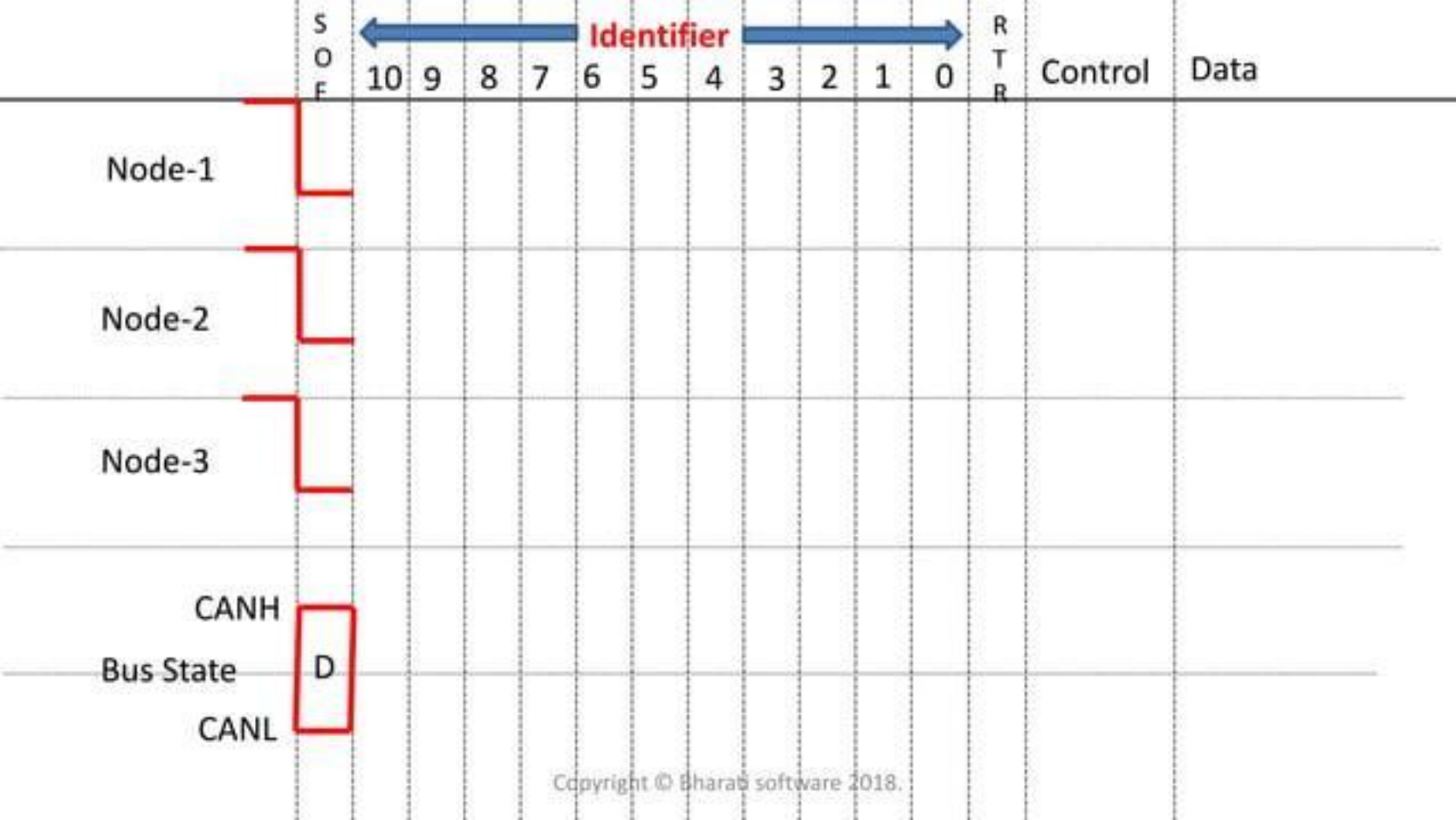
The CAN communication protocol is a carrier-sense, multiple-access protocol with collision detection and arbitration on message priority (CSMA/CD+AMP).

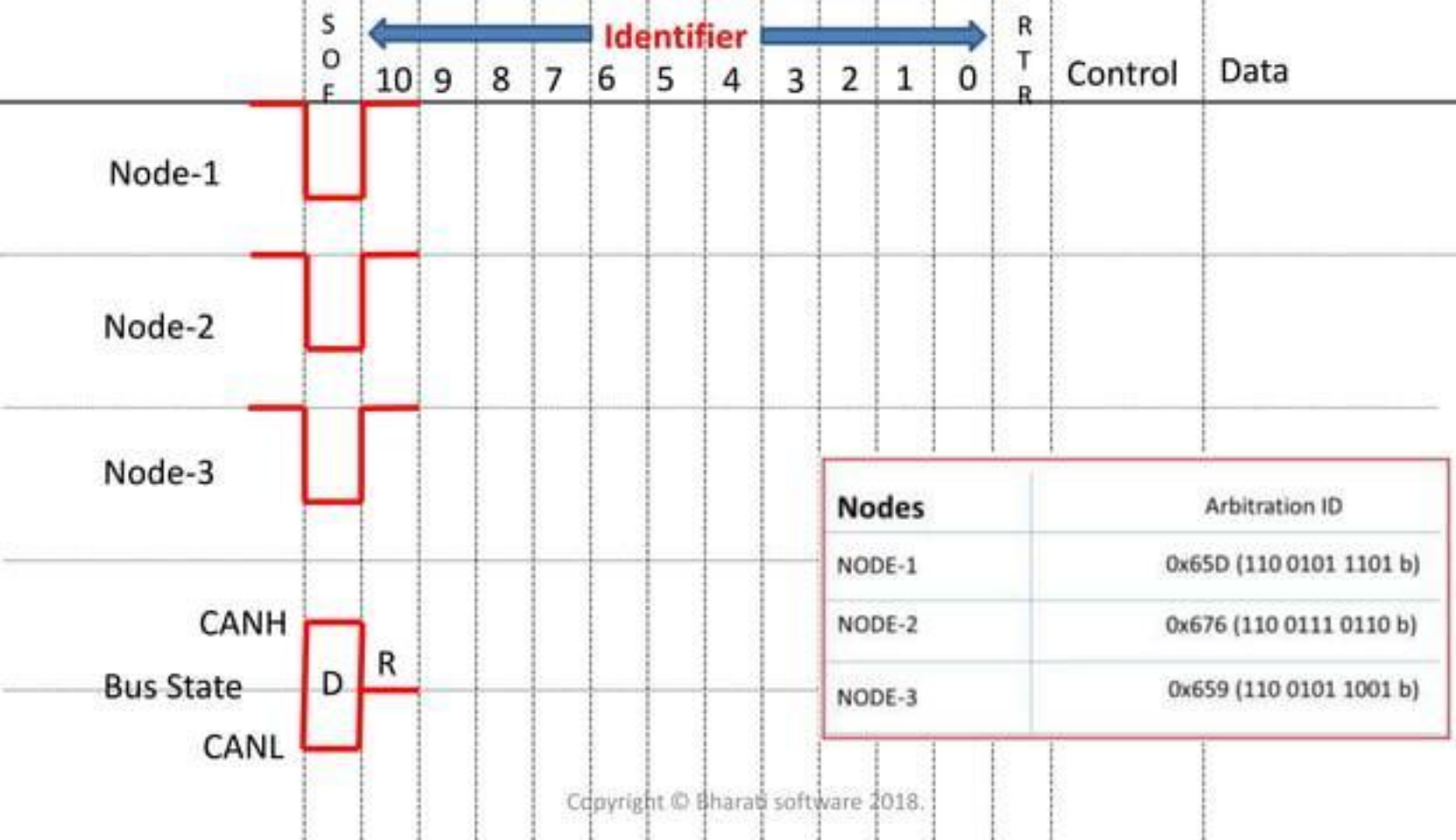
CSMA means that each node on a bus must wait for a prescribed period of inactivity before attempting to send a message. CD+AMP means that collisions are resolved through a bit-wise arbitration, based on a preprogramed priority of each message in the identifier field of a message

The higher priority identifier always wins bus access.

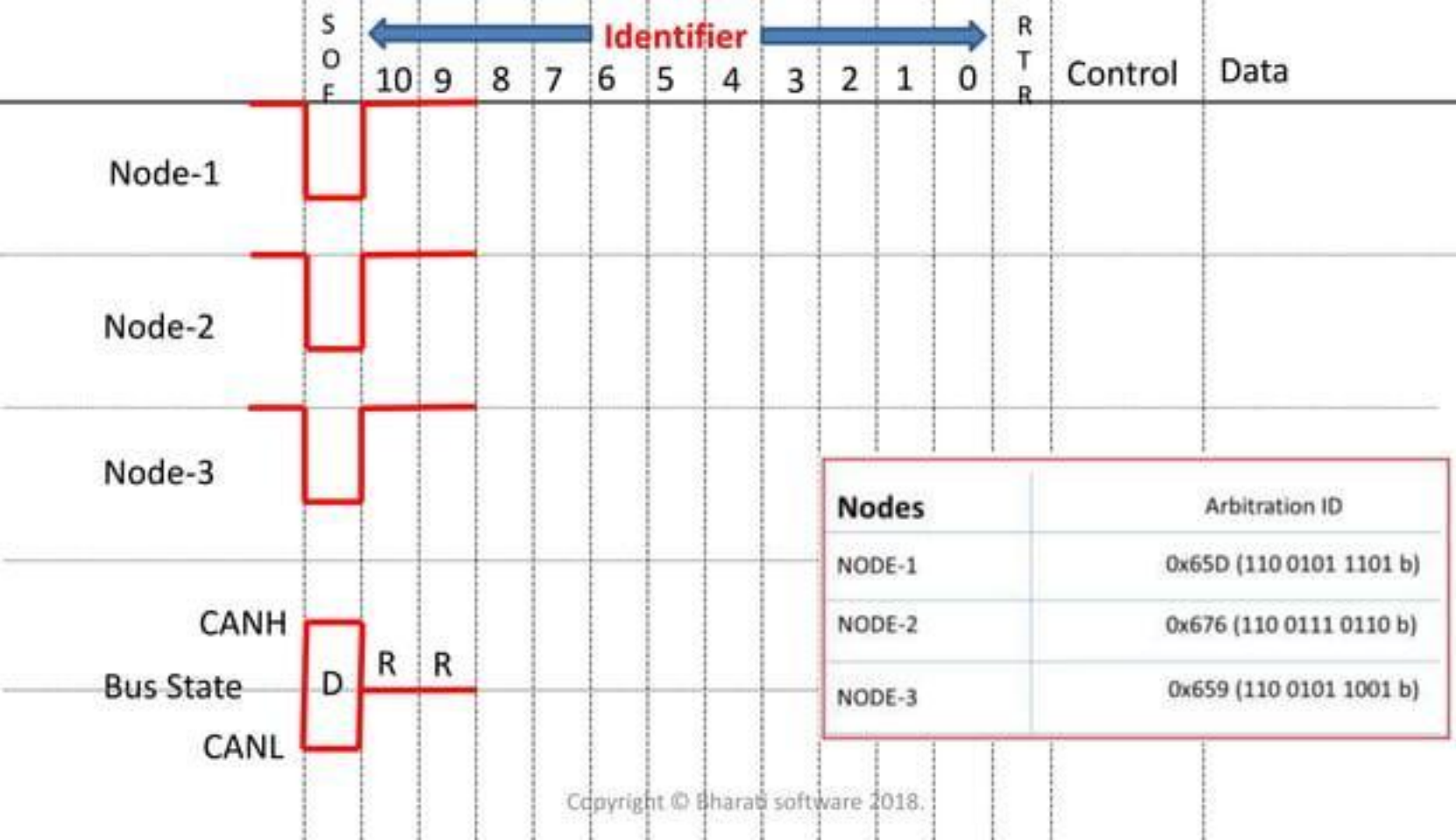
Bitwise Bus arbitration in CAN

Nodes	Arbitration ID
NODE-1	0x65D (110 0101 1101 b)
NODE-2	0x676 (110 0111 0110 b)
NODE-3	0x659 (110 0101 1001 b)

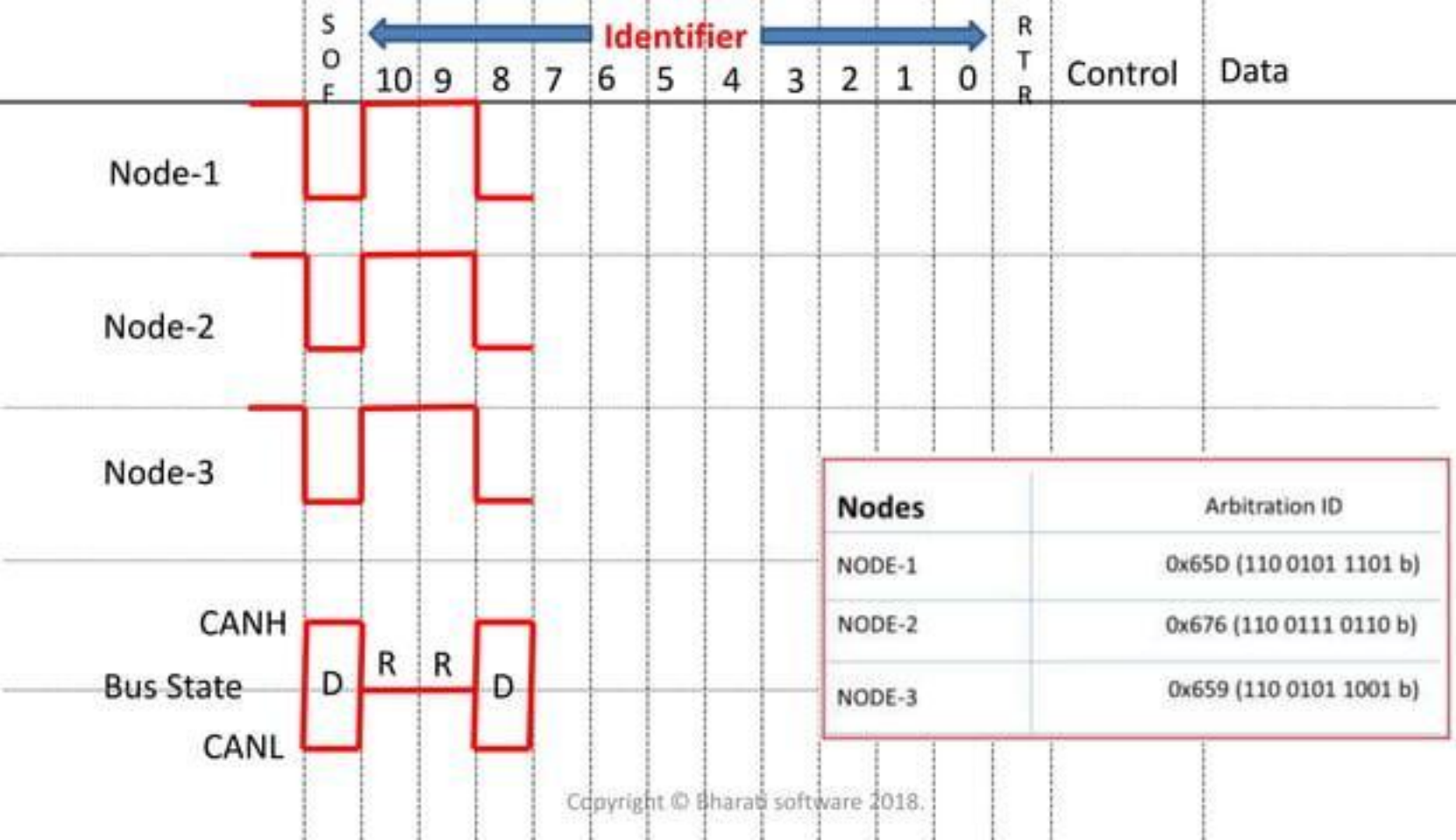




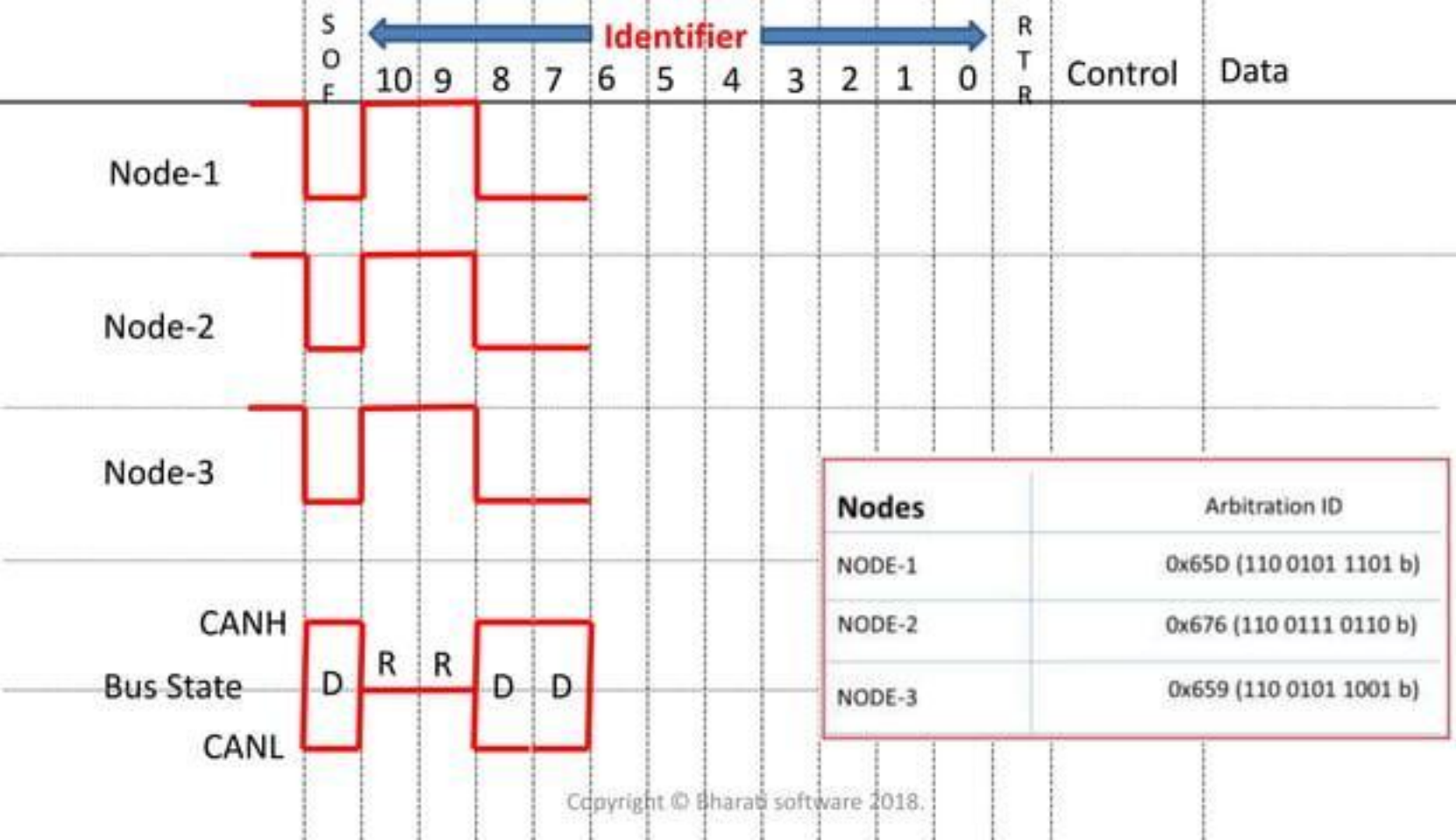
Nodes	Arbitration ID
NODE-1	0x65D (110 0101 1101 b)
NODE-2	0x676 (110 0111 0110 b)
NODE-3	0x659 (110 0101 1001 b)



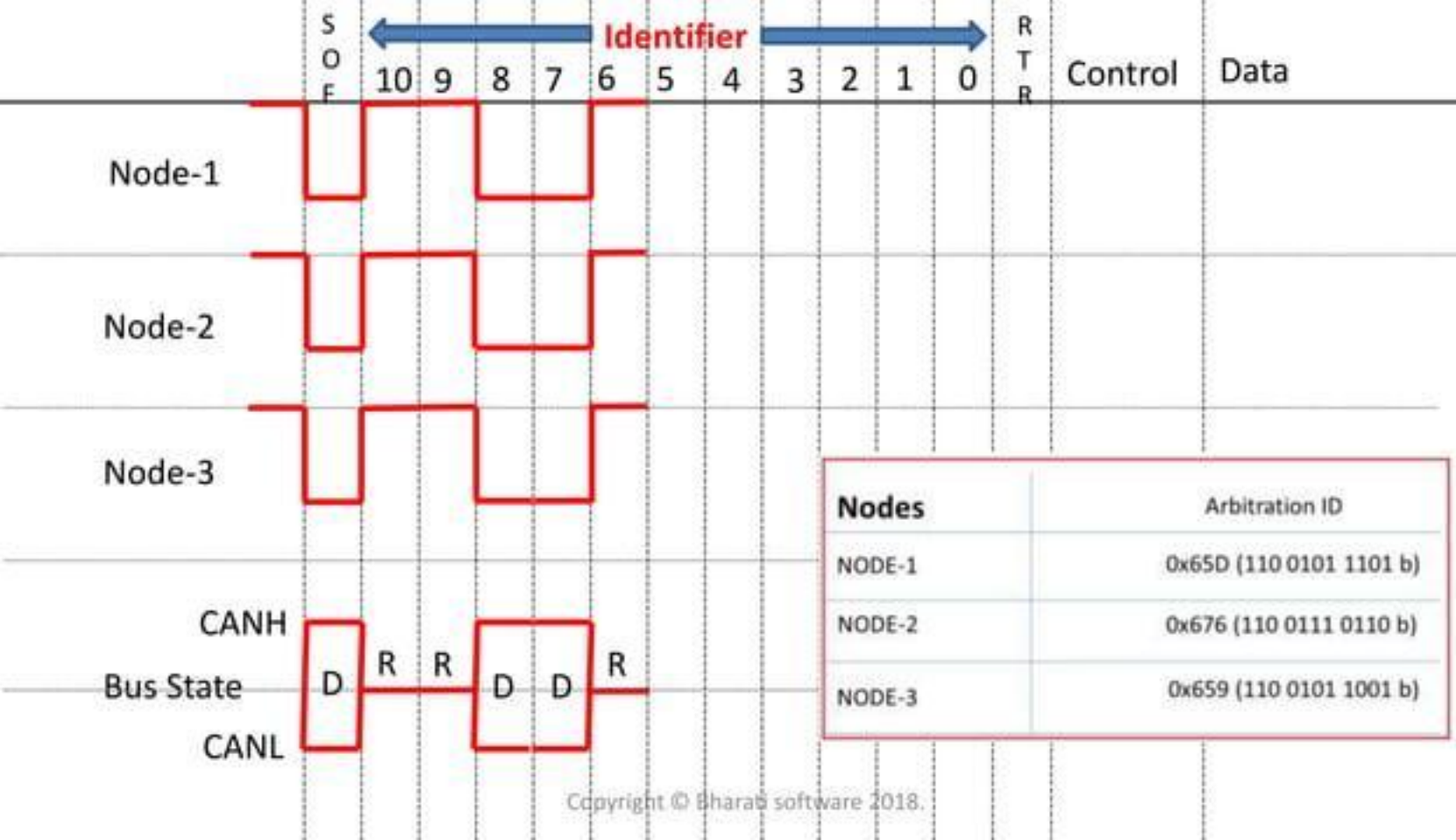
Nodes	Arbitration ID
NODE-1	0x65D (110 0101 1101 b)
NODE-2	0x676 (110 0111 0110 b)
NODE-3	0x659 (110 0101 1001 b)



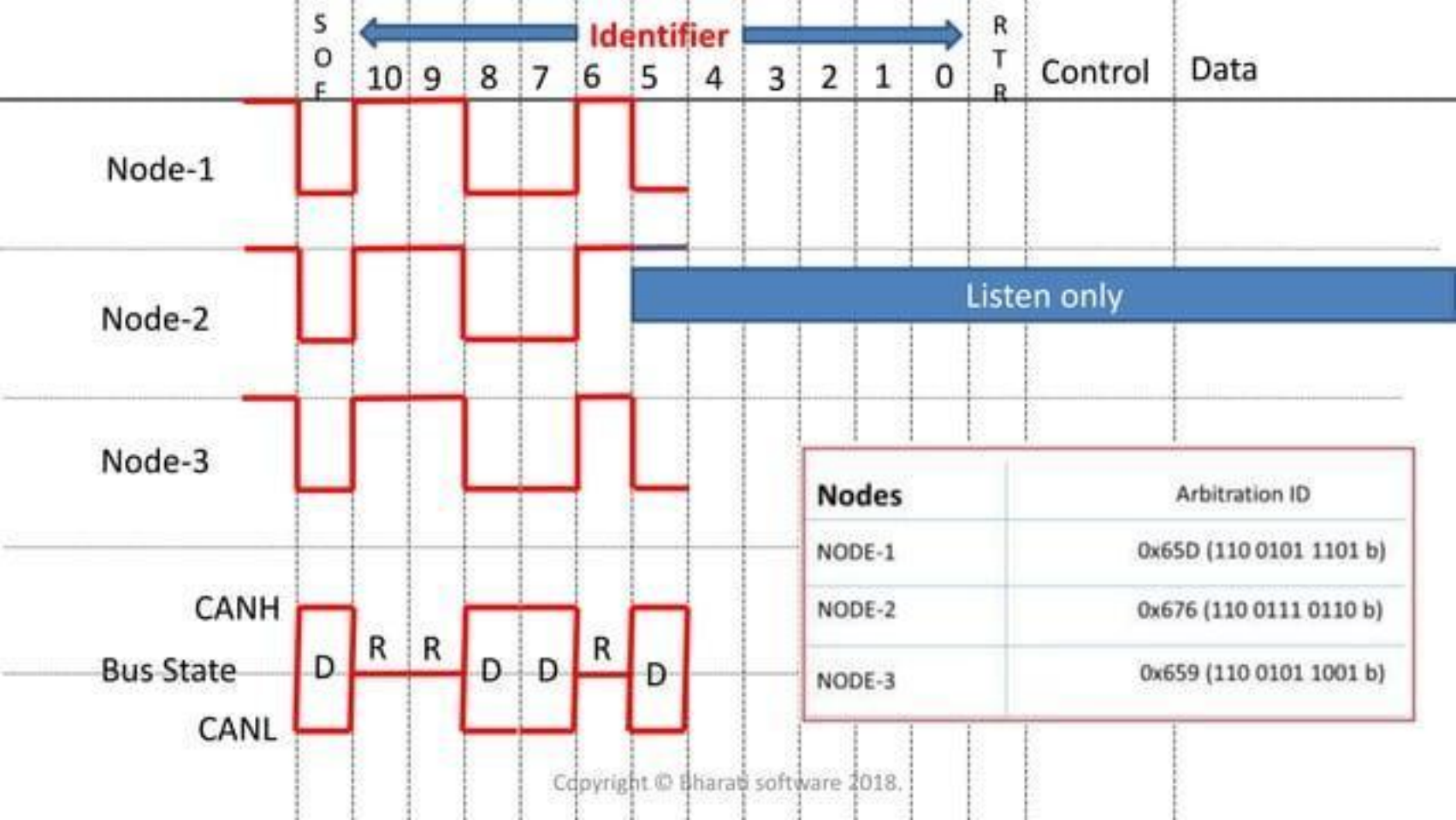
Nodes	Arbitration ID
NODE-1	0x65D (110 0101 1101 b)
NODE-2	0x676 (110 0111 0110 b)
NODE-3	0x659 (110 0101 1001 b)



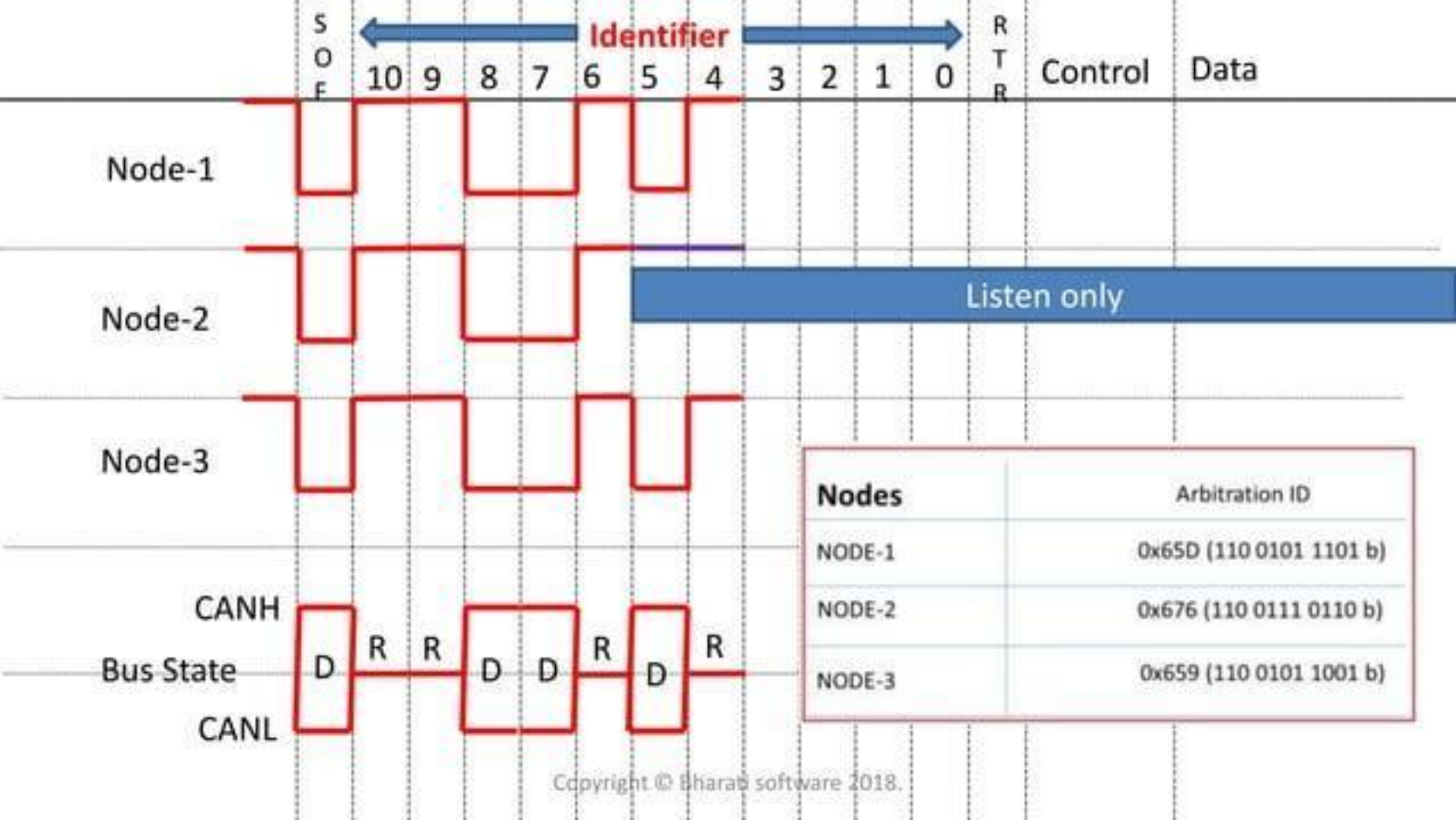
Nodes	Arbitration ID
NODE-1	0x65D (110 0101 1101 b)
NODE-2	0x676 (110 0111 0110 b)
NODE-3	0x659 (110 0101 1001 b)



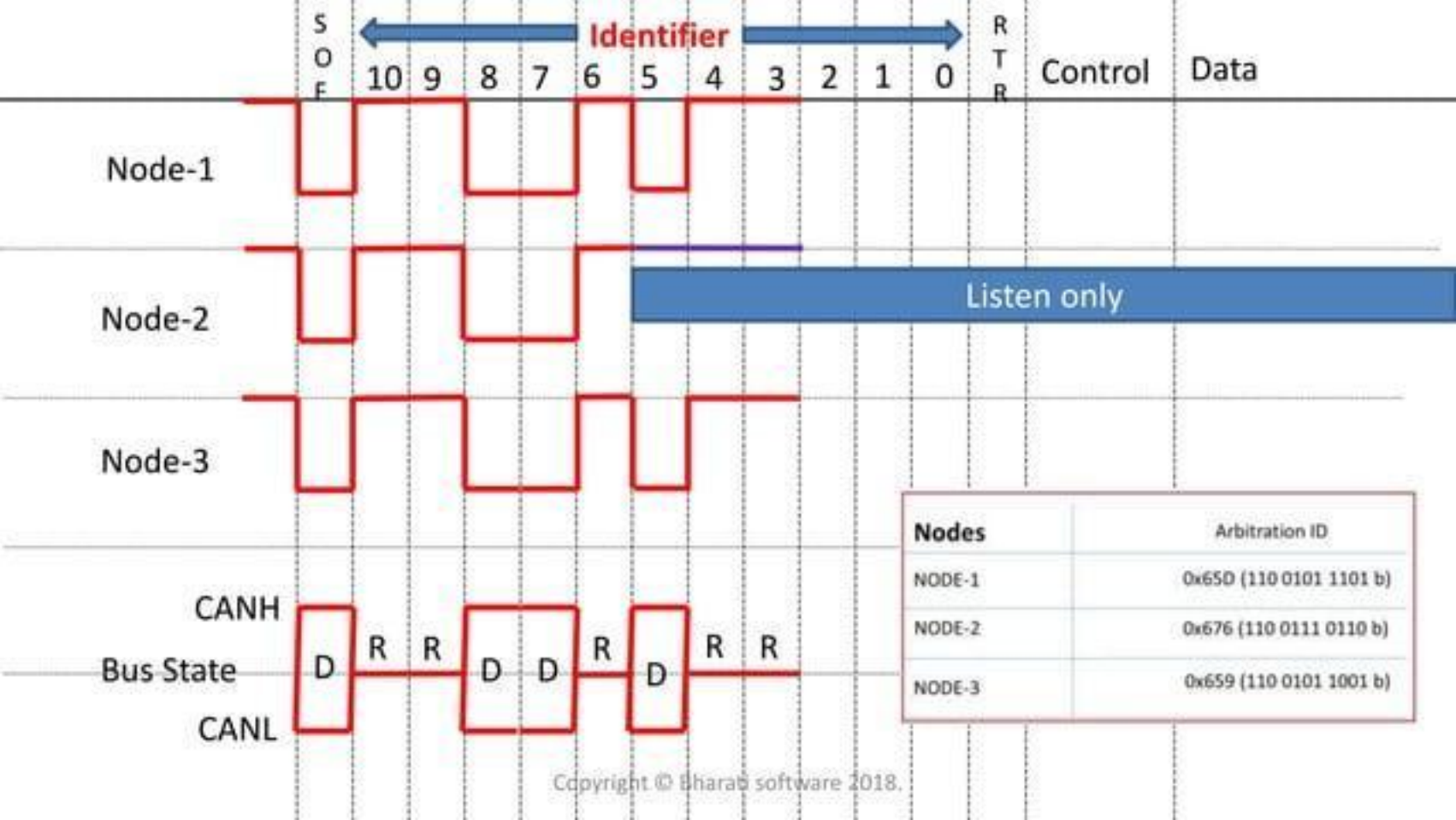
Nodes	Arbitration ID
NODE-1	0x65D (110 0101 1101 b)
NODE-2	0x676 (110 0111 0110 b)
NODE-3	0x659 (110 0101 1001 b)



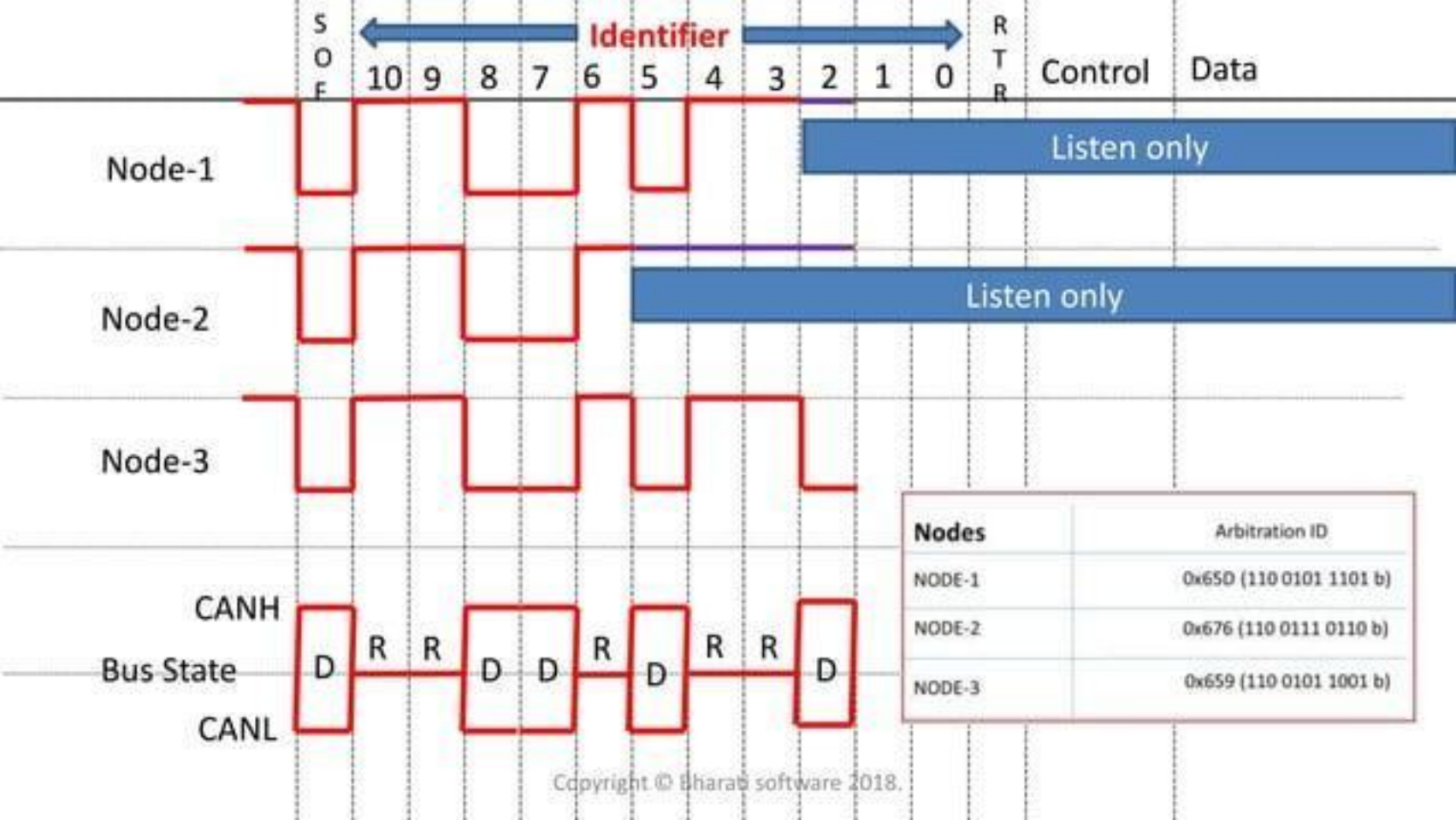
Nodes	Arbitration ID
NODE-1	0x65D (110 0101 1101 b)
NODE-2	0x676 (110 0111 0110 b)
NODE-3	0x659 (110 0101 1001 b)



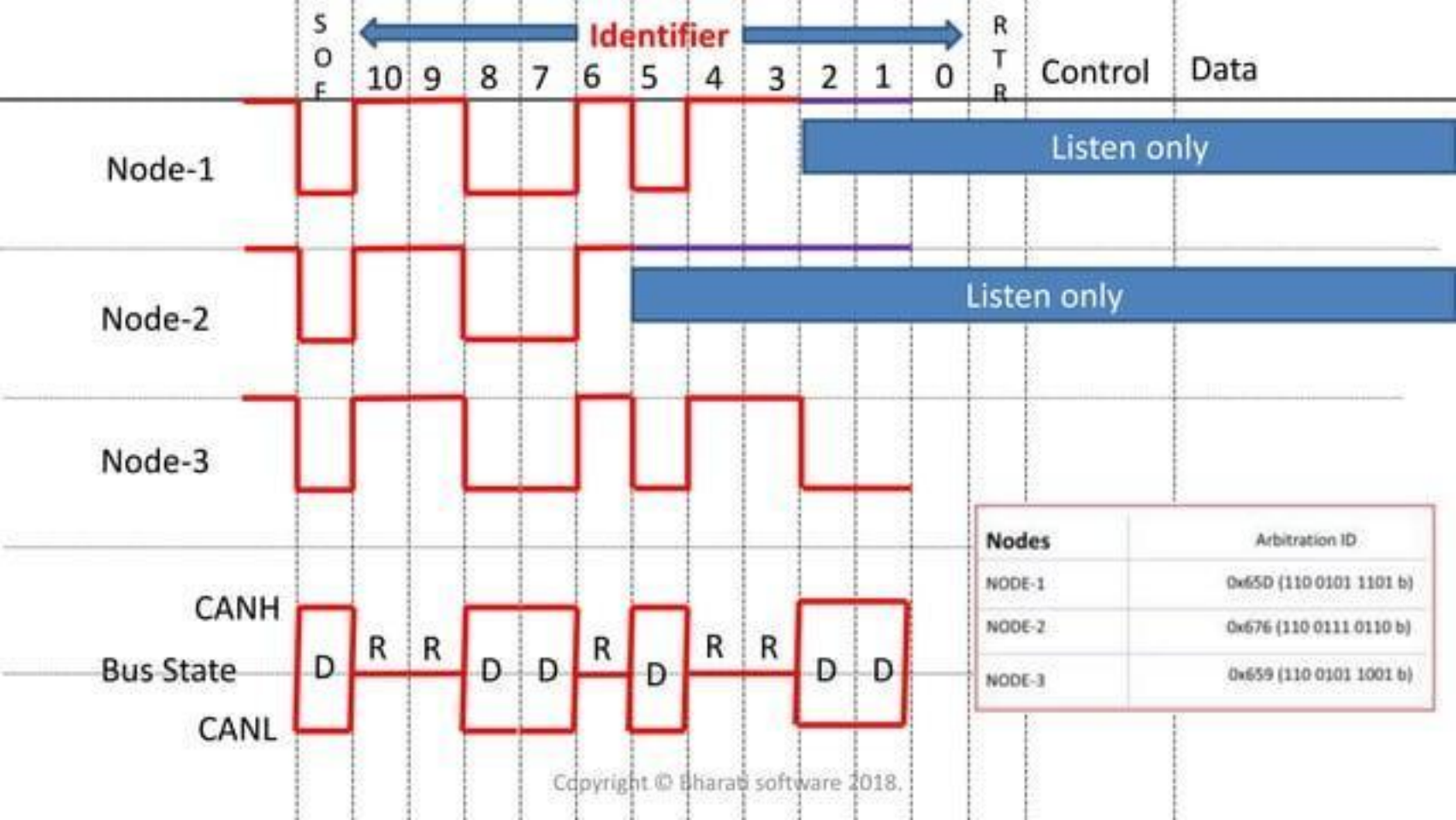
Nodes	Arbitration ID
NODE-1	0x65D (110 0101 1101 b)
NODE-2	0x676 (110 0111 0110 b)
NODE-3	0x659 (110 0101 1001 b)



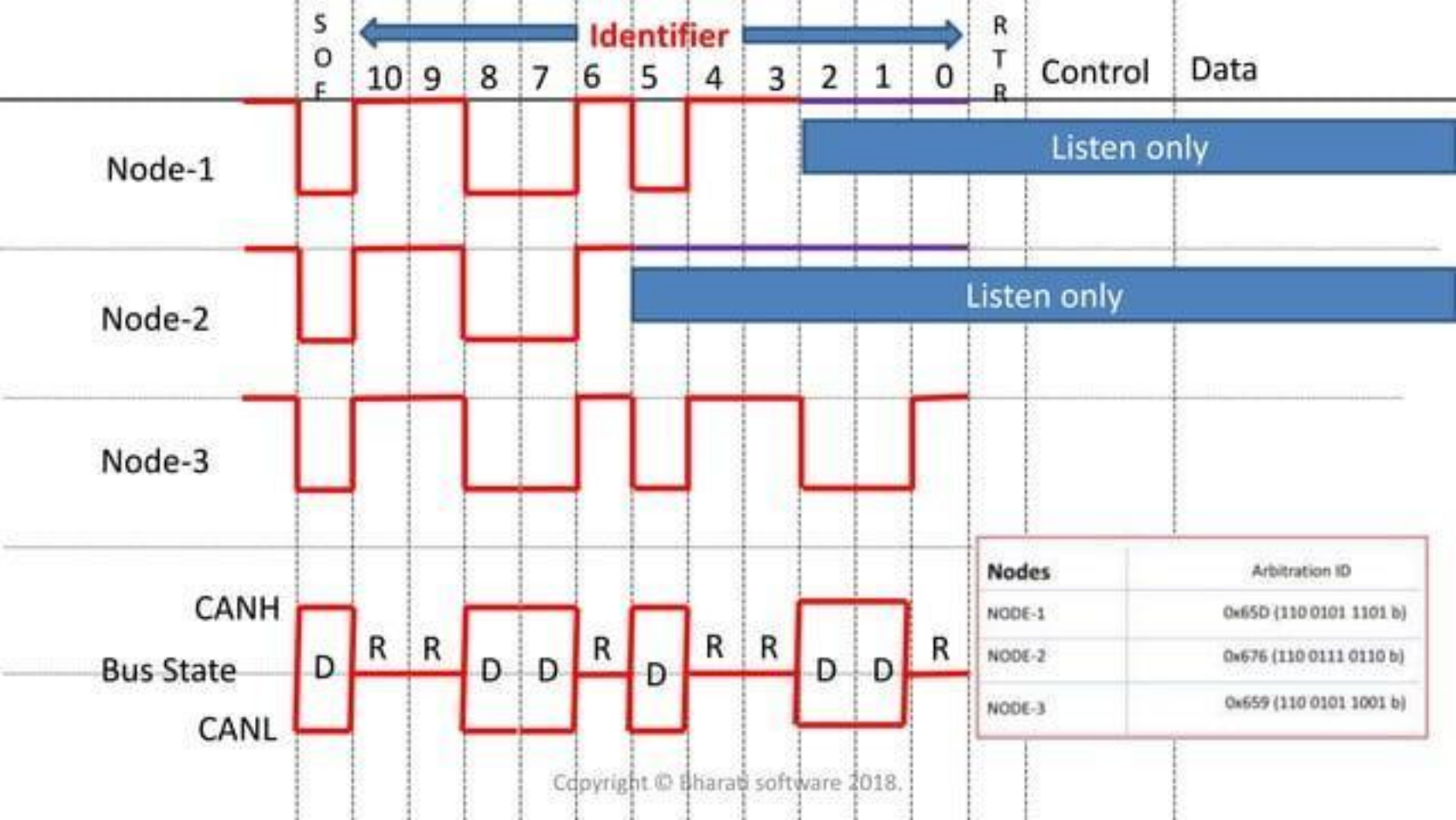
Nodes	Arbitration ID
NODE-1	0x65D (110 0101 1101 b)
NODE-2	0x676 (110 0111 0110 b)
NODE-3	0x659 (110 0101 1001 b)



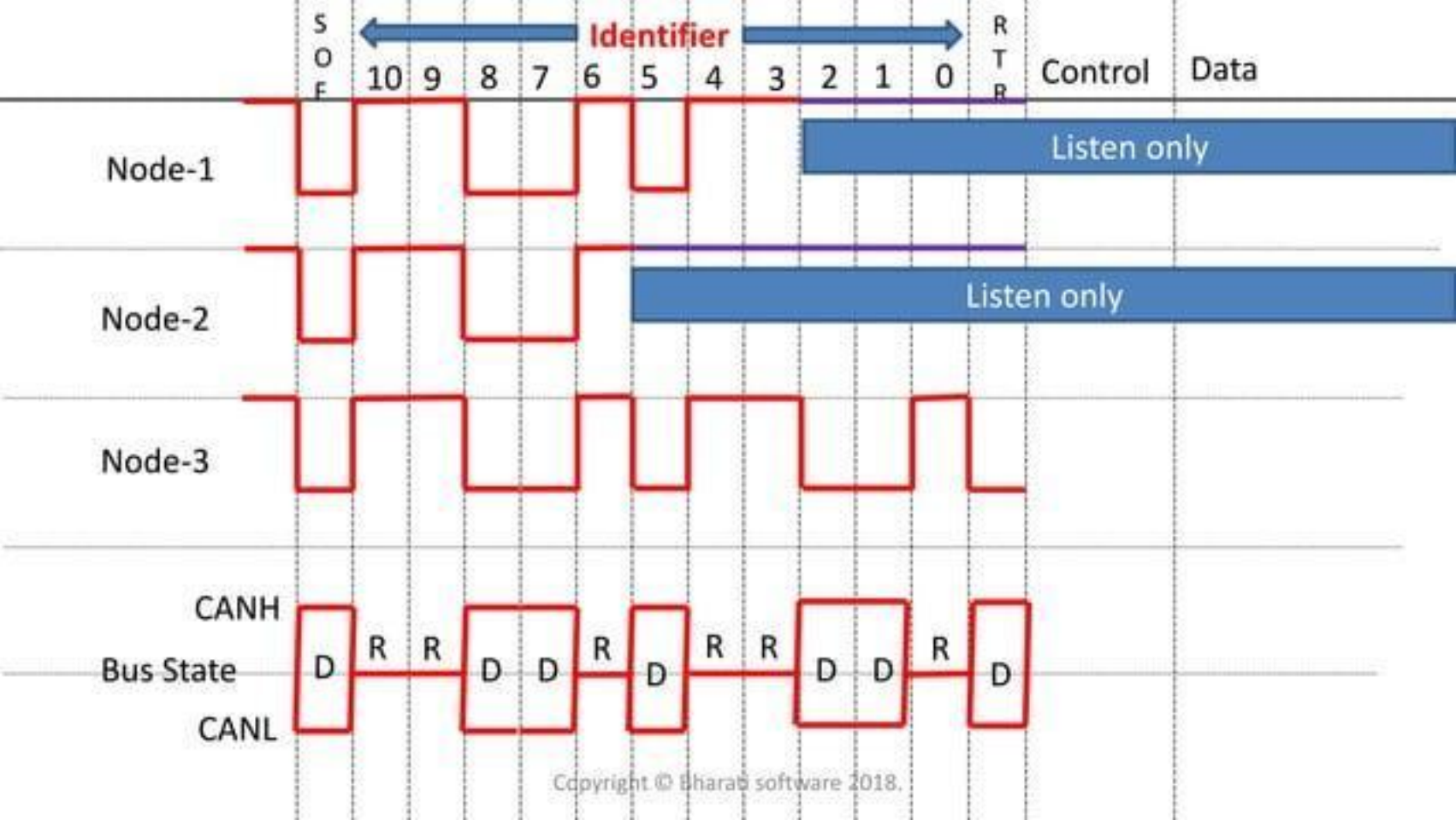
Nodes	Arbitration ID
NODE-1	0x65D (110 0101 1101 b)
NODE-2	0x676 (110 0111 0110 b)
NODE-3	0x659 (110 0101 1001 b)

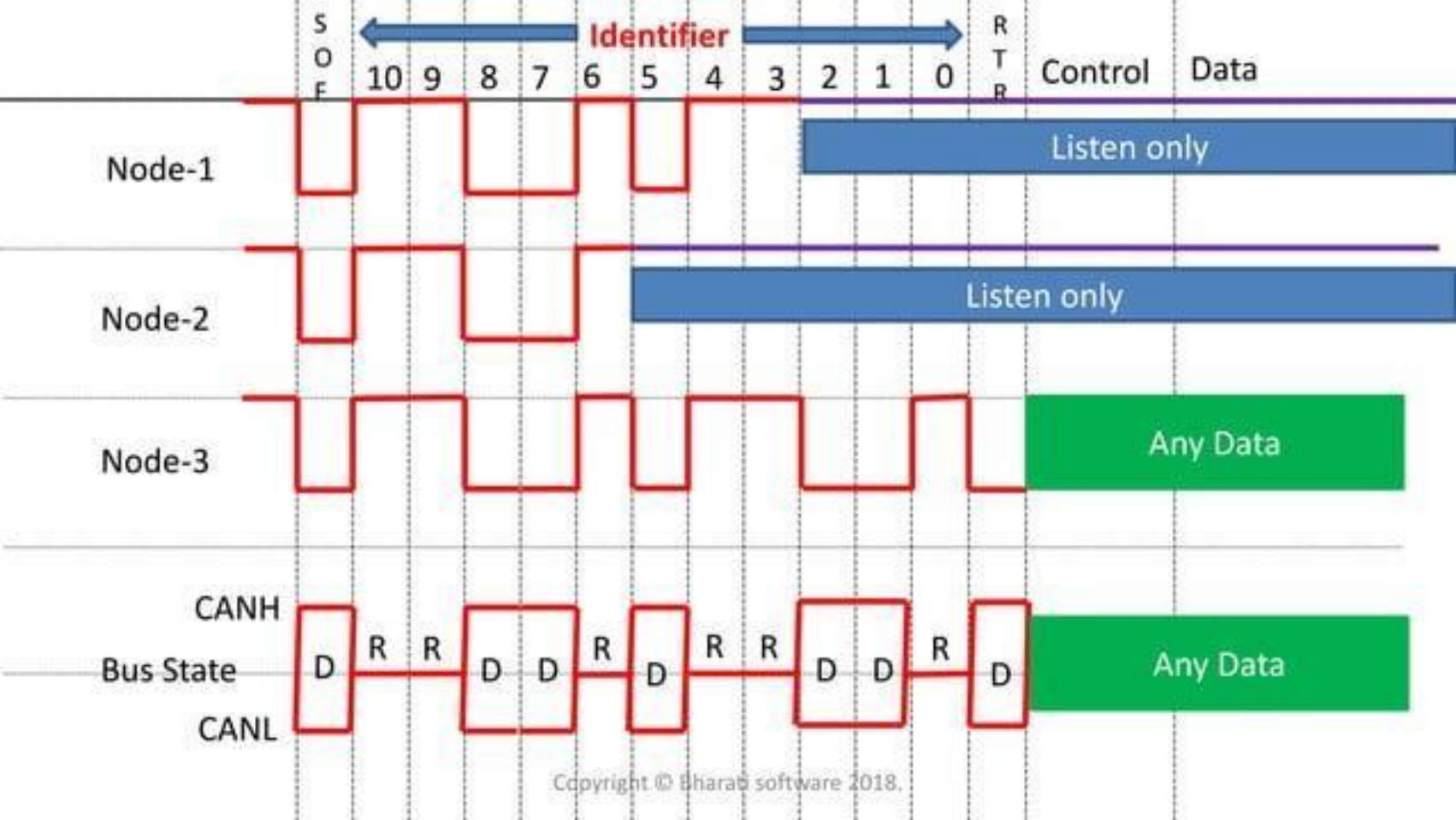


Nodes	Arbitration ID
NODE-1	0x65D (110 0101 1101 b)
NODE-2	0x676 (110 0111 0110 b)
NODE-3	0x659 (110 0101 1001 b)



Nodes	Arbitration ID
NODE-1	0x65D (110 0101 1101 b)
NODE-2	0x676 (110 0111 0110 b)
NODE-3	0x659 (110 0101 1001 b)


SOF
Identifier
10 9 8 7 6 5 4 3 2 1 0
RTR
Control
Data
Node-1
Listen only
Node-2
Listen only
Node-3
CANH
Bus State
CANL
D R R D D R D R R D D R D
Copyright © Bharab software 2018.



ST's BxCAN Controller

The bxCAN(Basic xtended CAN) module handles the transmission and the reception of CAN messages fully autonomously. Standard identifiers (11-bit) and extended identifiers (29-bit) are fully supported by the hardware

ST's bxCAN features

- 2 CAN Controllers are available CAN1 and CAN2
- CAN1 is called Master bxCAN and CAN2 is slave
- Both Supports CAN protocol version 2.0 A, B
- Bit rates up to 1 Mbit/s
- Three transmit mailboxes
- Two receive FIFOs with three stages
- 28 filter banks shared between CAN1 and CAN2

What application can do with bxCAN ?

- Configure CAN parameters, e.g. bit rate, bit timings , etc
- Transmissions
- Handle receptions
- Manage interrupts
- Get diagnostic information

bxCAN block Diagram

Refer to RM : Figure 383. Dual CAN block diagram

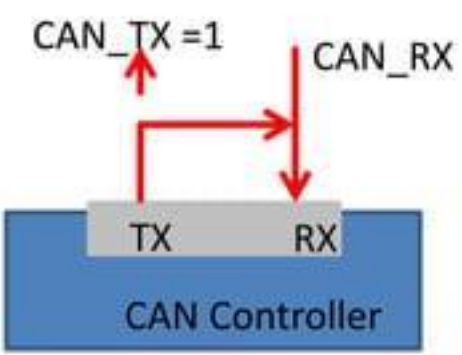
bxCAN Master and Slave

As mentioned in reference manual RM0390

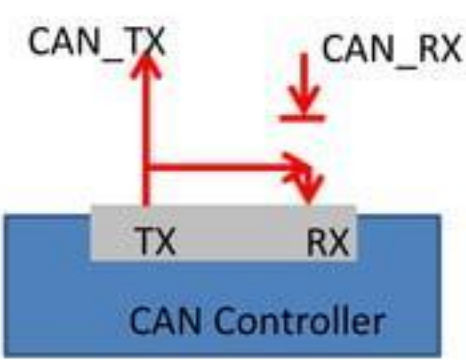
"- CAN1: Master bxCAN for managing the communication between a Slave bxCAN and the 512-byte SRAM memory.

-CAN2: Slave bxCAN, with no direct access to the SRAM memory."

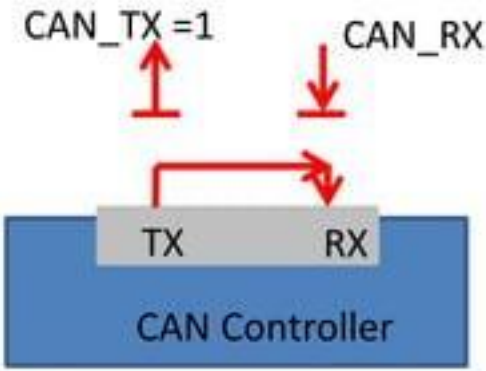
BxCAN Test Modes



Silent Mode

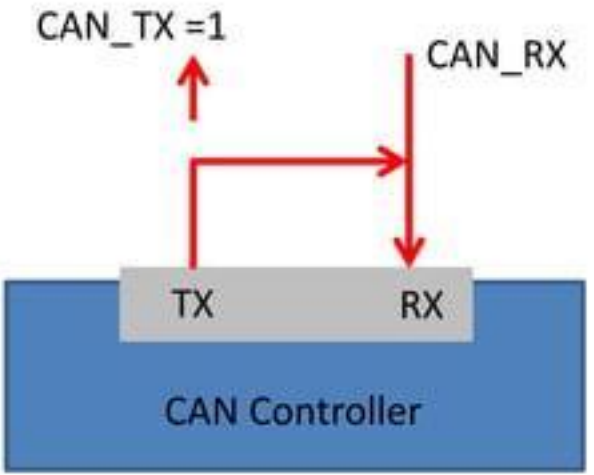


Loop Back Mode



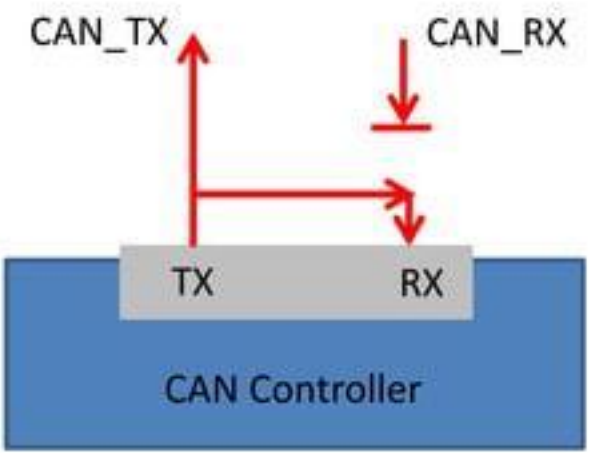
Silent+ Loop Back Mode

Silent Mode



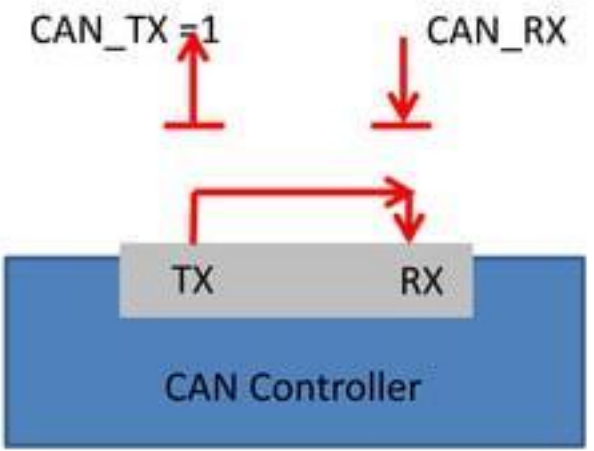
- Tx line is internally looped back to RX line
- CAN TX is held at recessive state
- bxCAN is able to receive valid frames
- It just listens and doesn't change the bus state by putting dominant bit
- Can be used as a sniffer which just analyzes the traffic on the bus.

Loop Back Mode



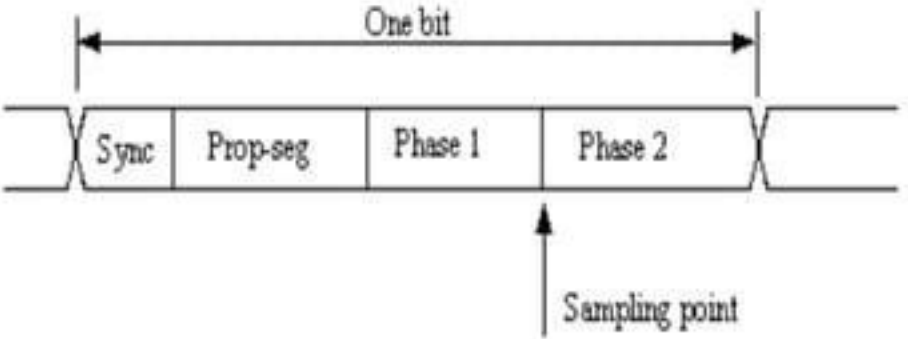
- bxCAN can Transmit frames on the bus .
- Also the frames are looped back to the RX line internally
- bxCAN will not listen to the bus, but just receives its own message which is looped back.
- loop back mode is provided for self test functions

Silent Look Back Mode



- bxCAN controller is totally disconnected from the bus
- It neither transmits nor listens to the bus
- TX is internally looped back to the RX, hence receives its own messages.

CAN Bit Timings Configuration



Each bit on the CAN bus is, for timing purposes, divided in to 4 segments

1) Synchronization Segment
2) Propagation Segment
3) Phase Segment 1
4) Phase Segment 2

Width of these segments have to be adjusted properly to get desired bit rate on the CAN bus. Width of each segment is mentioned in terms of time quanta

The Time Quanta is the smallest time unit for all configuration values.

bXCAN Block Diagram(TX-Path)

- Three transmit mailboxes are provided to the software for setting up messages.
- The transmission Scheduler decides which mailbox has to be transmitted first.
- In order to transmit a message, the application must select one **empty** transmit mailbox, set up the identifier, the data length code (DLC) and the data before requesting the transmission
- Request Transmission by setting TXRQ bit in the control register.
- Immediately after the TXRQ bit has been set, the mailbox enters **pending** state and waits to become the highest priority mailbox

bXCAN Block Diagram(TX-Path)

- As soon as the mailbox has the highest priority it will be **scheduled** for transmission.
- The transmission of the message of the scheduled mailbox will start (enter **transmit** state) when the CAN bus becomes idle
- Once the mailbox has been successfully transmitted, it will become **empty** again
- The hardware indicates a successful transmission by setting the RQCP and TXOK bits in the CAN_TSR register.
- If the transmission fails, the cause is indicated by the ALST bit in the CAN_TSR register in case of an Arbitration Lost, and/or the TERR bit, in case of transmission error detection

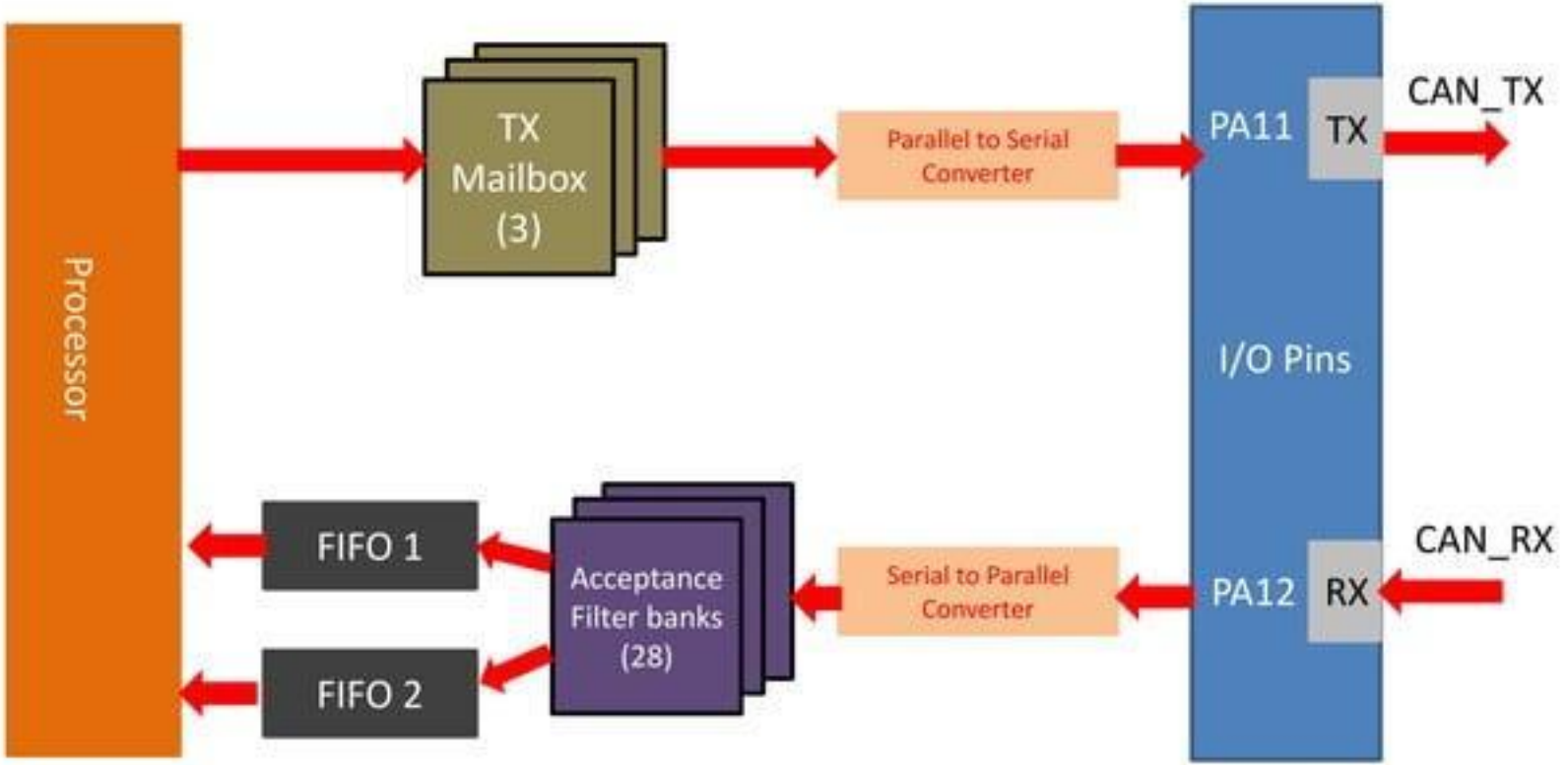
bxCAN Tx and STM32 Cube

```
  * @brief  CAN Tx message header structure definition
  */
typedef struct
{
  uint32_t StdId;    /*!< Specifies the standard identifier.
                         This parameter must be a number between Min_Data = 0 and Max_Data = 0x7FF. */

  uint32_t ExtId;    /*!< Specifies the extended identifier.
                         This parameter must be a number between Min_Data = 0 and Max_Data = 0x1FFFFFFF. */

  uint32_t IDE;      /*!< Specifies the type of identifier for the message that will be transmitted.
                         This parameter can be a value of @ref CAN_identifier_type */

  uint32_t RTR;      /*!< Specifies the type of frame for the message that will be transmitted.
                         This parameter can be a value of @ref CAN_remote_transmission_request */

  uint32_t DLC;      /*!< Specifies the length of the frame that will be transmitted.
                         This parameter must be a number between Min_Data = 0 and Max_Data = 8. */

  FunctionalState TransmitGlobalTime; /*!< Specifies whether the timestamp counter value captured on start
                         of frame transmission, is sent in DATA6 and DATA7 replacing pData[6] and pData[7].
                         @note: Time Triggered Communication Mode must be enabled.
                         @note: DLC must be programmed as 8 bytes, in order these 2 bytes are sent.
                         This parameter can be set to ENABLE or DISABLE. */

} CAN_TxHeaderTypeDef;
```

bxCAN Tx and STM32 Cube

```
HAL_CAN_AddTxMessage(CAN_HandleTypeDef *hcan,
                     CAN_TxHeaderTypeDef *pHeader,
                     uint8_t aData[], uint32_t *pTxMailbox)
```

bXCAN Block Diagram(RX-Path)



bxCAN RX Path

- Two receive FIFOs are used by each CAN Controller to store the incoming messages
- Three complete messages can be stored in each FIFO
- The FIFOs are managed completely by hardware

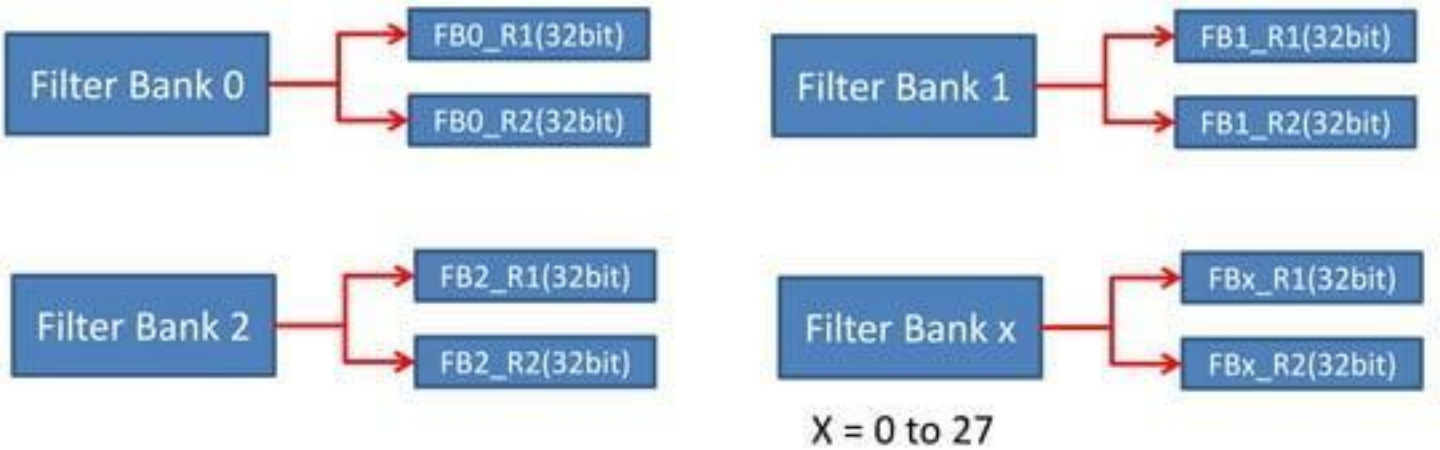
RX Filtering

- The controller will read any frames it sees on the bus and hold them in a small FIFO memory. It will notify the host processor that this data is available which the processor then reads from the controller
- The controller also contains a hardware filter mechanism that can be programmed to ignore and discard those CAN frames you do not want passed to the processor. This saves processor overhead.
- Acceptance filtering is introduced to manage the frame reception

Acceptance filtering

- There are 28 filter banks shared between Master bxCAN (CAN1) and slave bxCAN(CAN2)
- Each Filter bank has 2 , 32 bit associated filter registers.
- You can use filter banks to filter the incoming messages. Lets see some examples.

Filter Banks



Frame acceptance Rules: Example

1. Accept frames only if first 3 msbs of the standard identifier are 1s
   e.g. 111xxxx

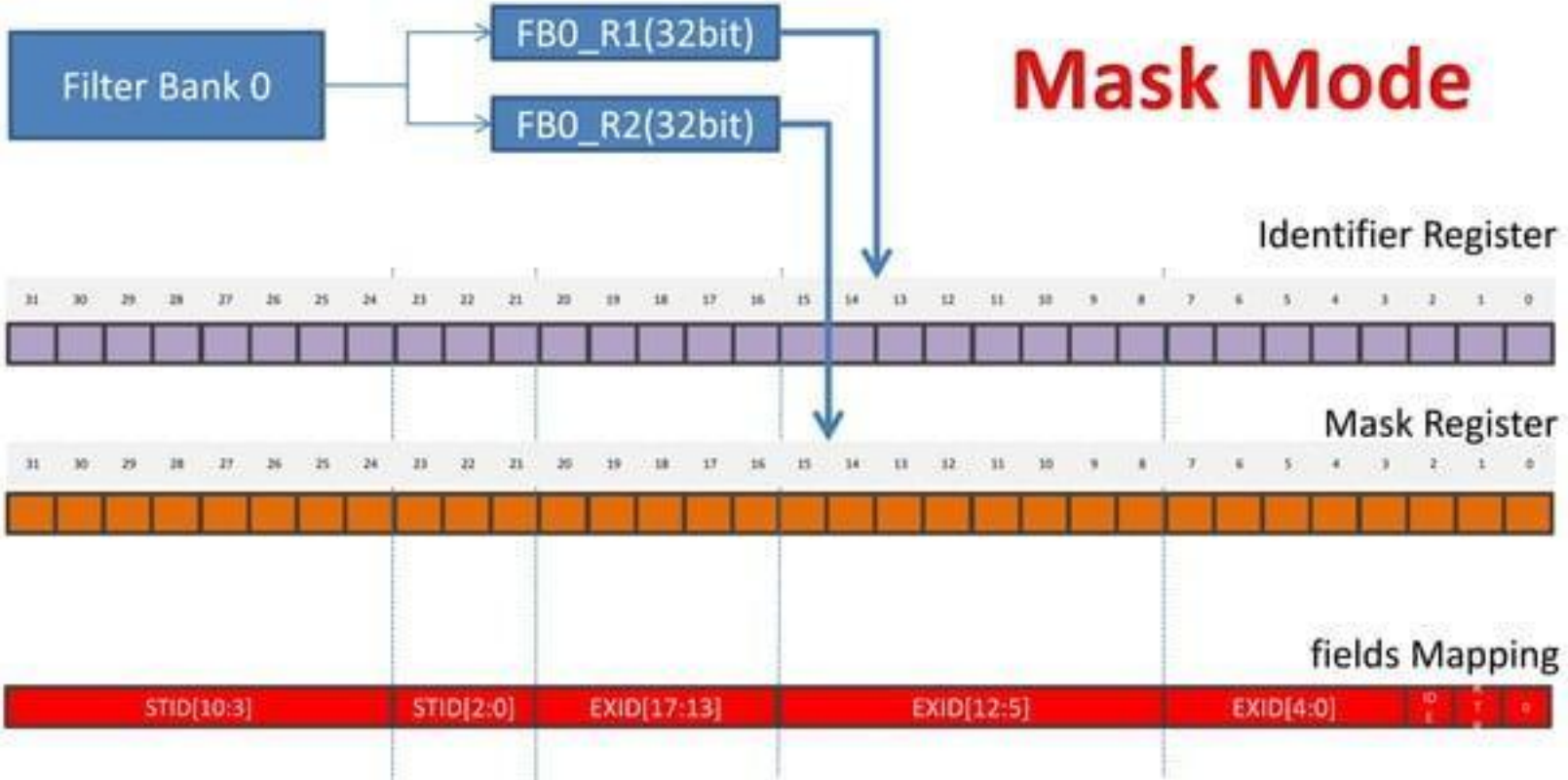
2. Accept frames only if first 3 msbs of the standard identifer are 0s and last 2 lsbs are 1s

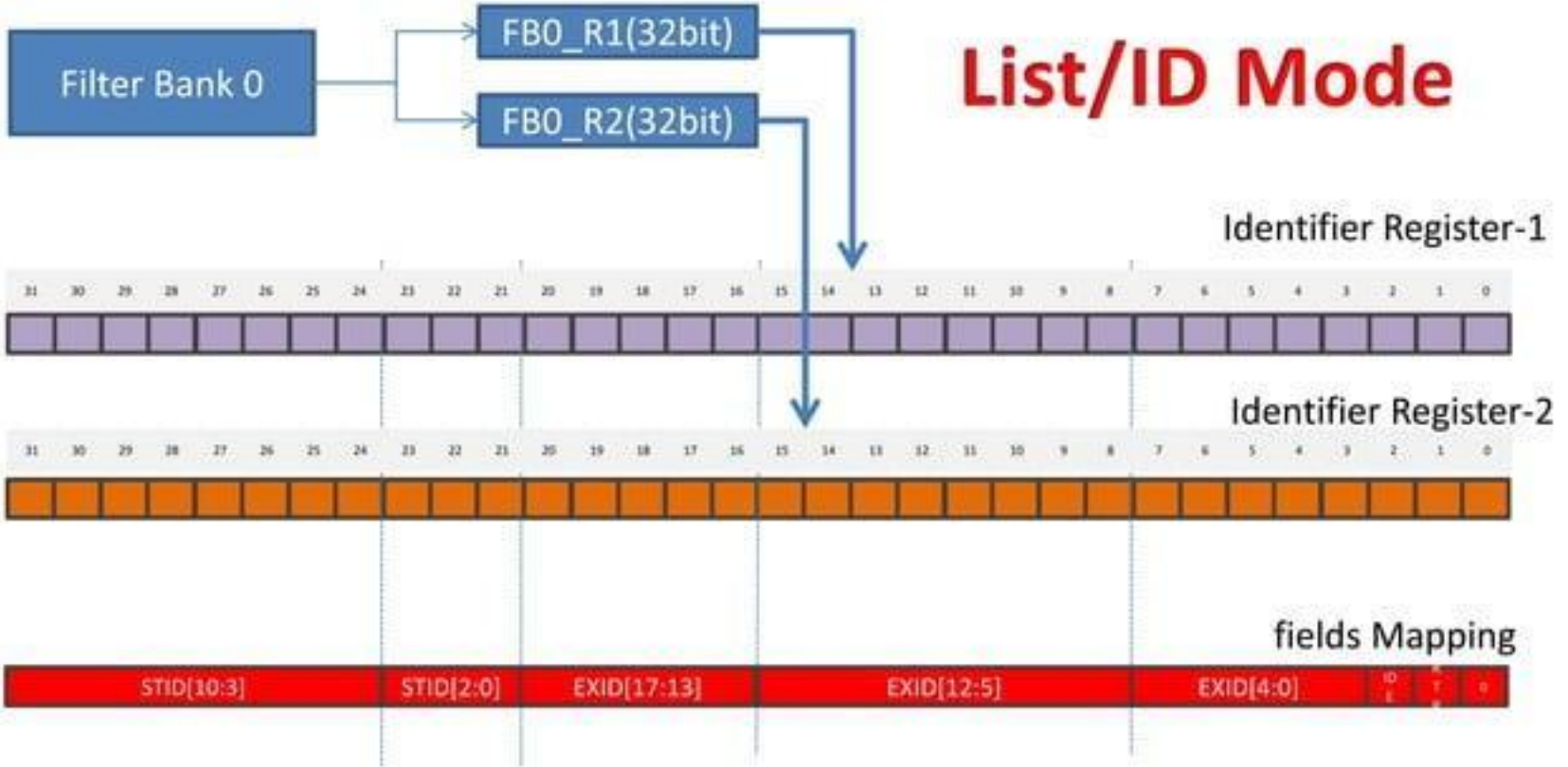
3. Accept frames only if standard identifier value exactly = 0x65D or 0x651

4. Accept only Request frames

5. Accept only Extended Id Frames

6. Accept all frames





Receive FIFO State Machine

Refer RM : Figure 389. Receive FIFO states

bxCAN Interrupts

When does bxCAN issue interrupts ?

1. Transmit Request Completed
2. Frame is received in RX FIFO0
3. Frame is received in RX FIFO1
4. During CAN status change or Error

Four interrupt vectors(IRQs) are dedicated to bxCAN1
Four interrupt Vectors(IRQs) are dedicated to bxCAN2

Exercise

Find out the bxCAN1 and bxCAN2 IRQ numbers for STM32F446RE microcontroller

Hint : Refer Vector table information in RM

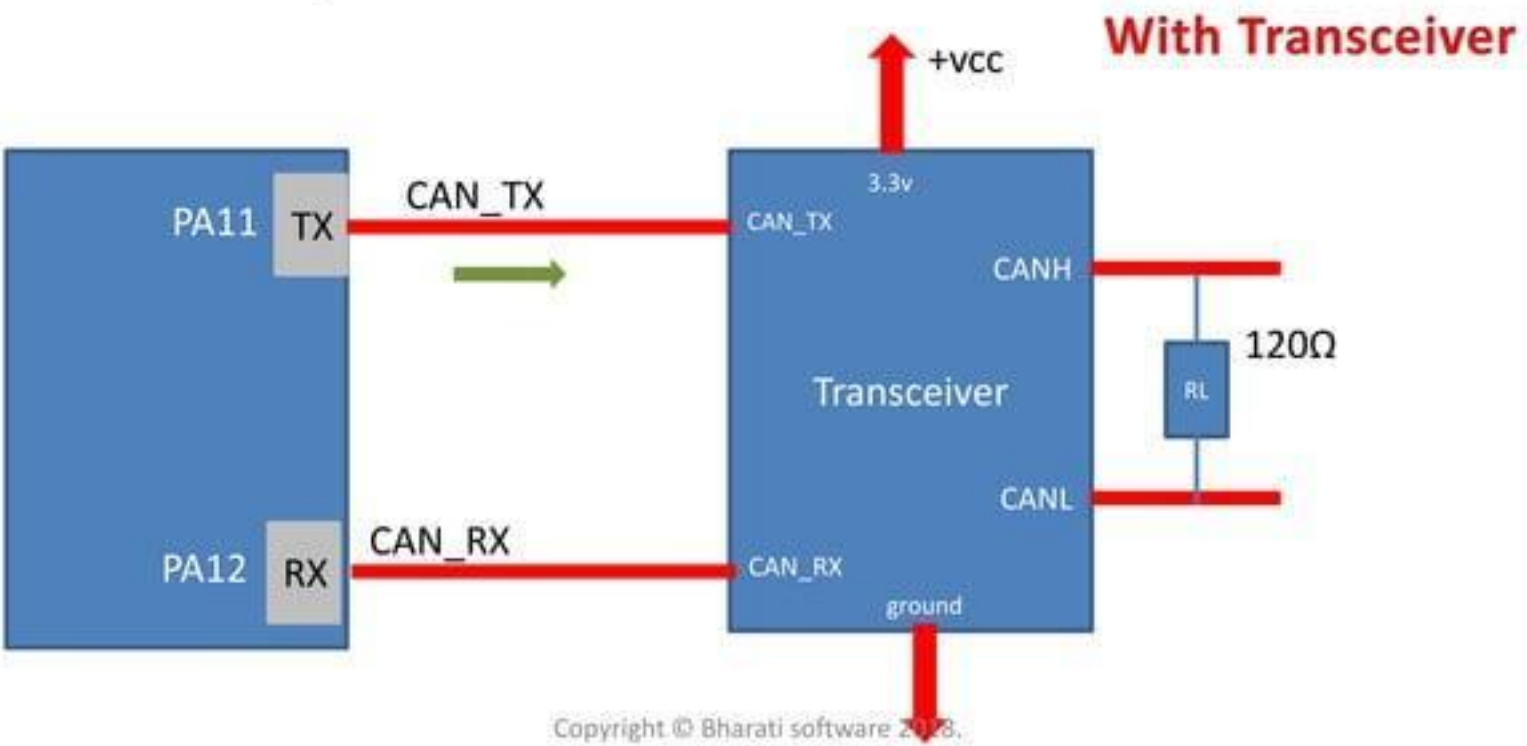
bxCAN interrupt generation

Refer RM : Figure 396. Event flags and interrupt generation

Reception related interrupts

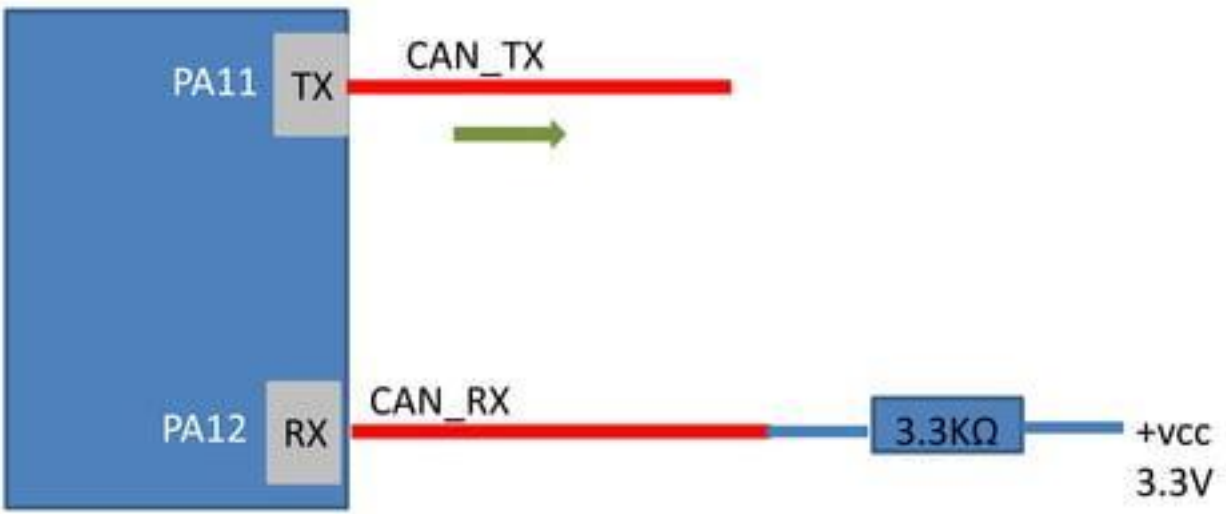
- Once a message has been stored in the FIFO
- interrupt request is generated
- When the FIFO becomes full interrupt is generated
- On overrun condition

Loop Back mode connection

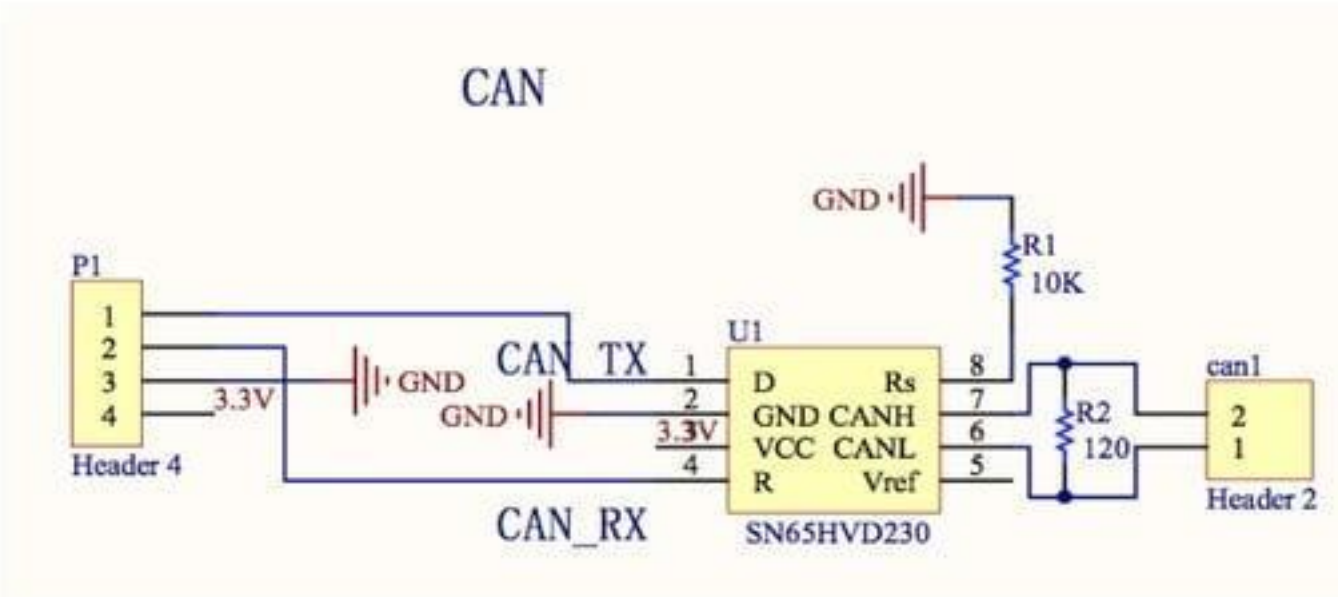


Loop Back mode connection

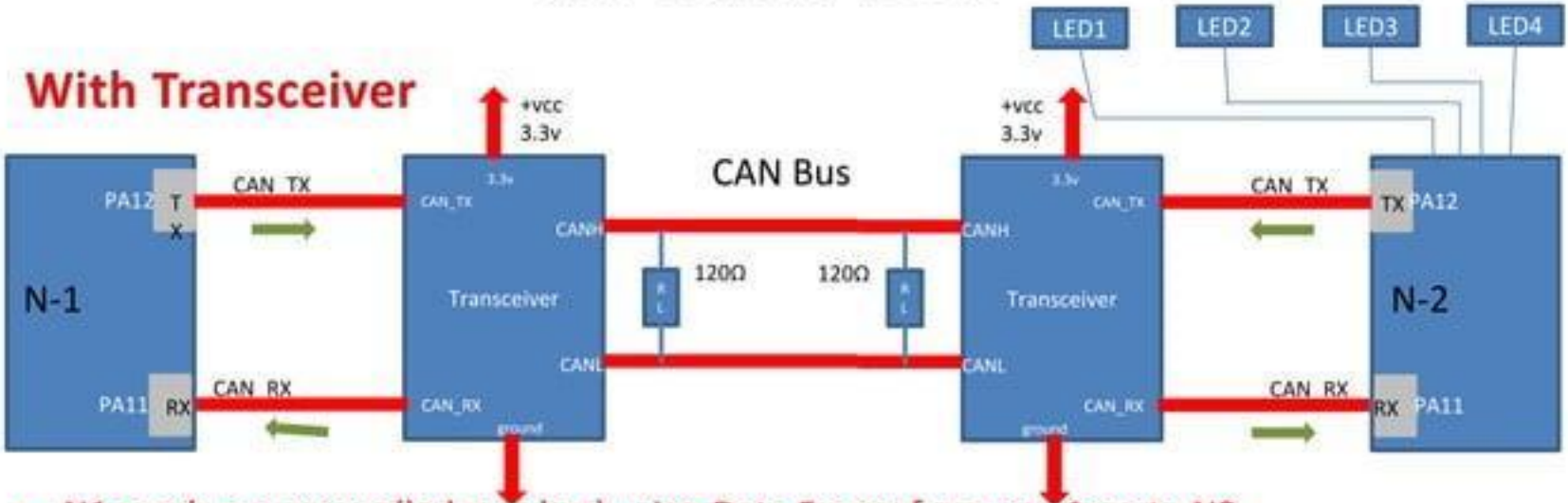
Without Transceiver



SN65HVD230 CAN Transceiver Schematic

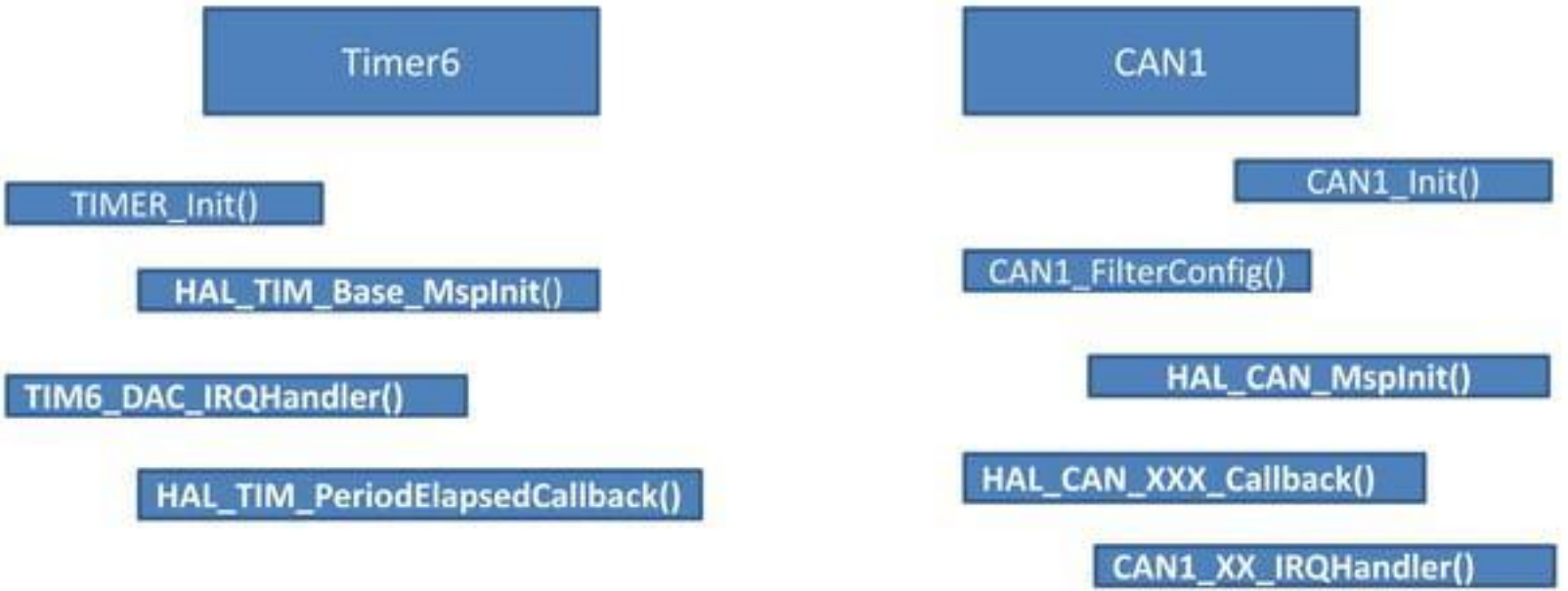


CAN Normal Mode

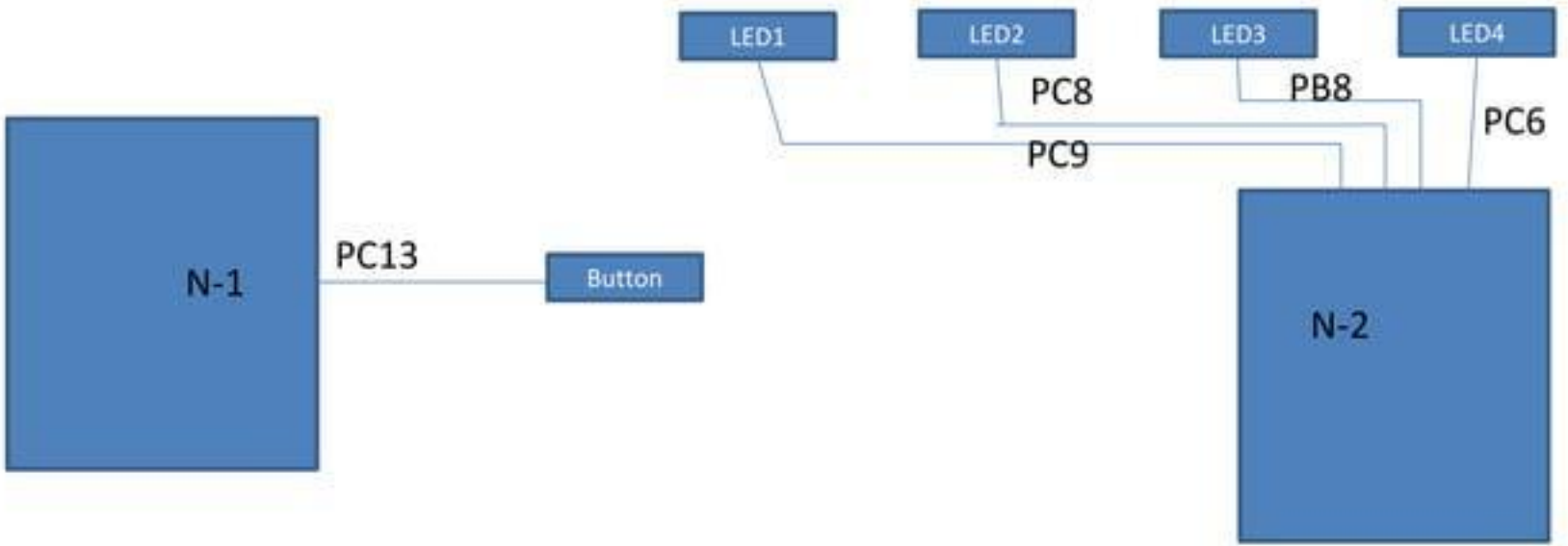


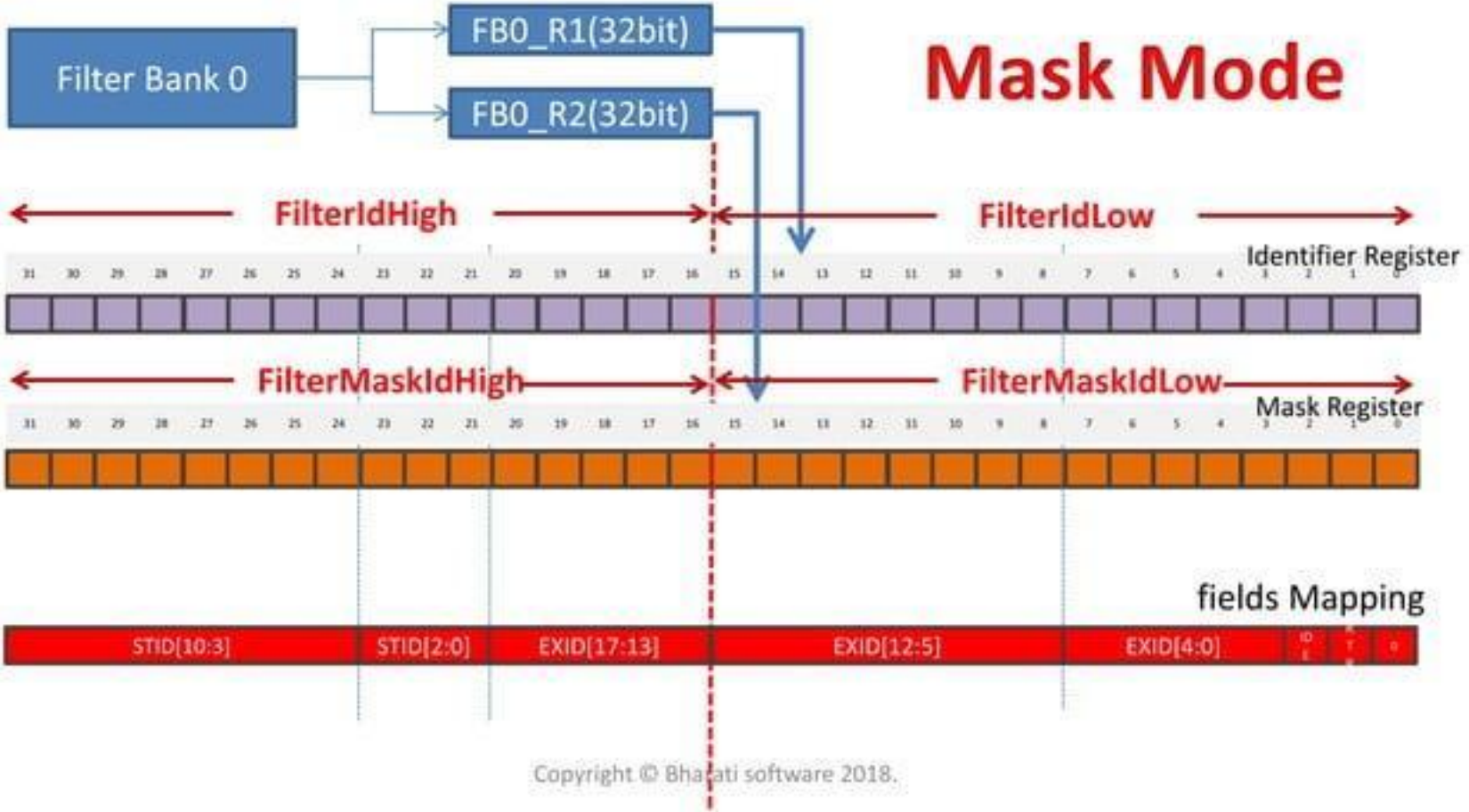
- N1 sends a message(led number) using Data Frame for every 1sec to N2
- After the reception of message(led number) N2 has to glow the corresponding LED
- N1 also sends a Remote frame to request 2 bytes of data for every 4sec
- N2 upon receiving the Remote frame should send back 2 bytes of data using Data Frame
- Use Interrupt driven Code

CAN Normal Mode Exercise



N1 and N2 GPIO Inits





Low Power Modes

MCU Low power modes

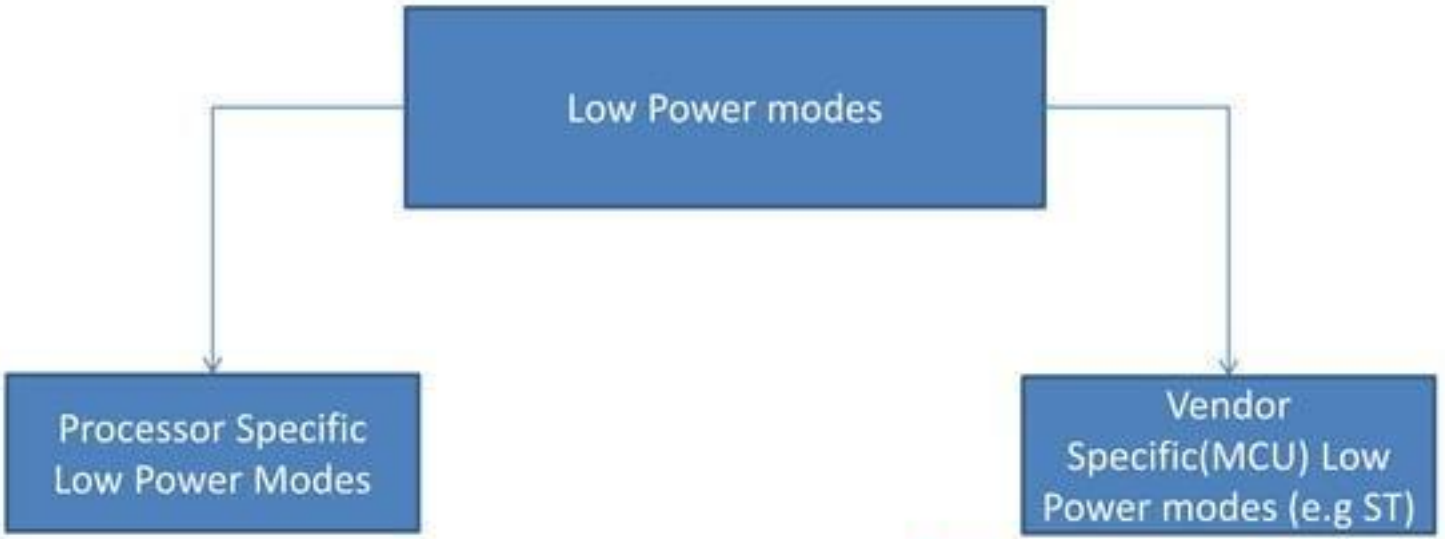
- A number of low-power features are available in the Cortex-Mx processor.
- In addition, microcontroller vendors usually also implement a number of low power modes in their Cortex-Mx-based microcontrollers. This we call as device specific low power modes or features .
- Details for microcontroller-specific low-power features are usually available in user manuals or application notes available from the microcontroller vendor Web sites

MCU Low Power Modes

A Microcontroller can be in Run mode or in Low Power mode.

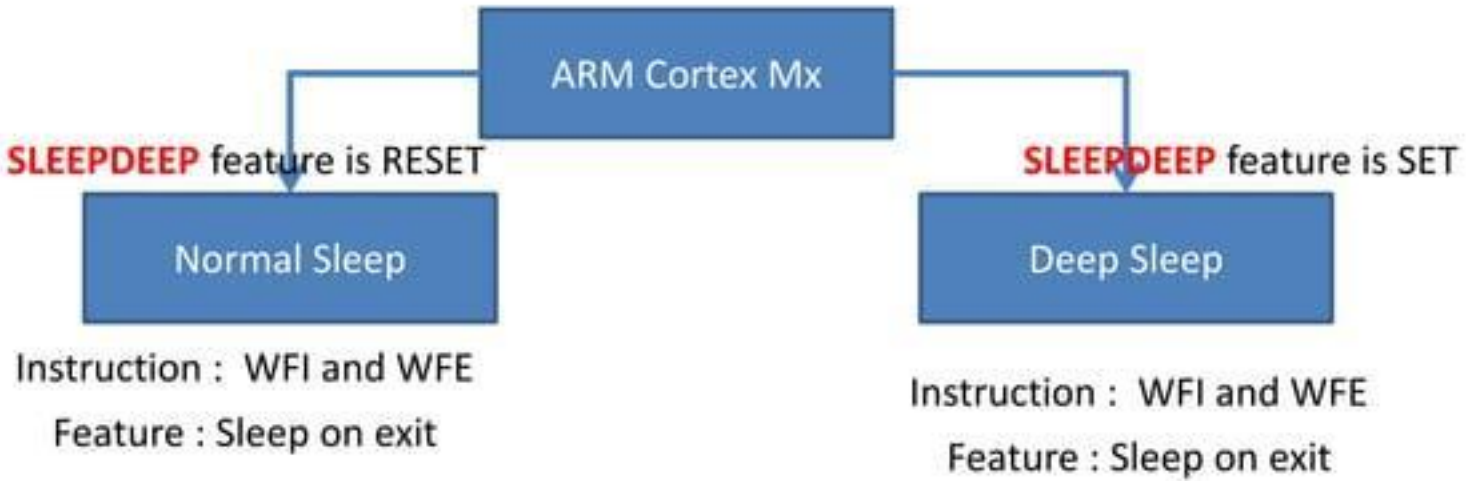
In Run mode the processor will be clocked always and it will be doing its normal operation and it will be consuming the power for its operation .

When processor has got nothing to do , you can send it to low power mode. Otherwise it will be in busy idle loop wasting CPU cycles thus consuming the power. For battery operated applications this is not a good design.



Additional sleep modes are introduced by MCU vendors using device-specific programmable registers

Processor Specific Low Power modes



Normal Sleep Vs Deep Sleep

- Inside the processor, the selection between normal sleep mode and deep sleep mode is defined by the **SLEEPDEEP** bit in the System Control Register(SCR)
- Normal sleep mode stops the processor clock
- Deep sleep mode stops the system clock and switches off the PLL and flash memory (*This is highly dependent on vendor who designs the microcontroller*)
- The Exact behavior of Normal and Deep Sleep mode is Microcontroller vendor specific .
- For more information about the behavior of the sleep modes see the documentation supplied by your device vendor.

How to Enter Normal Sleep

- SLEEPDEEP bit of the ARM Cortex Mx processor must be reset
- Use instructions like WFI or WFE to trigger entering the sleep mode
- You can also use SLEEPONEXIT feature of the ARM Cortex Mx processor to enter in to sleep mode.

How to Enter Deep Sleep

- SLEEPDEEP bit of the ARM Cortex Mx processor must be set
- Use instructions like WFI or WFE to trigger entering the sleep mode
- You can also use SLEEPONEXIT feature of the ARM Cortex Mx processor to enter in to sleep mode.

Summary : Entering Low Power Mode

In ARM cortex Mx based processor there are only 3 ways by which you can make processor enter in to the low power mode

- *execution of a WFE instruction*
- *execution of a WFI instruction*
- *using the Sleep-On-Exit feature*

If you enable the Sleep on Exit the processor enters sleep automatically when it exits ISR
Lets explore Sleep-On-Exit feature, WFI and WFE one by one

Entering SLEEP mode using Sleep-on-Exit feature

- ✓ Sleep on exit is a feature given by the ARM Cortex Mx processor .
- ✓ Remember its not an instruction
- ✓ When this feature is enabled, the processor automatically enters a sleep mode when exiting an exception handler if no other exception is waiting to be processed
- ✓ It does not cause the processor to enter sleep if the exception handler is returning to another exception handler (nested interrupt).

When to use this ?

When your application does all its work in an Interrupt handler , then while exiting ISR the Processor will automatically go to sleep if you enable this feature to save power .

- You would like to use this when processor runs only when an interrupt service require servicing.
- interrupt driven applications to stay in sleep mode as often as possible.
- The Sleep-On-Exit feature is ideal for interrupt-driven applications

Entering SLEEP mode using Sleep-on-Exit feature

- ✓ SLEEPONEXIT bit in the SCR of the ARM cortex Mx processor has to be set after all the initialization of your application
- ✓ No instruction is needed to enter sleep mode with this feature enabled
- ✓ When ISR finishes executing all the instructions you have written , processor goes in to sleep mode automatically without returning back to thread mode (No thread related un-stacking happens , because processor is not going back to thread mode)

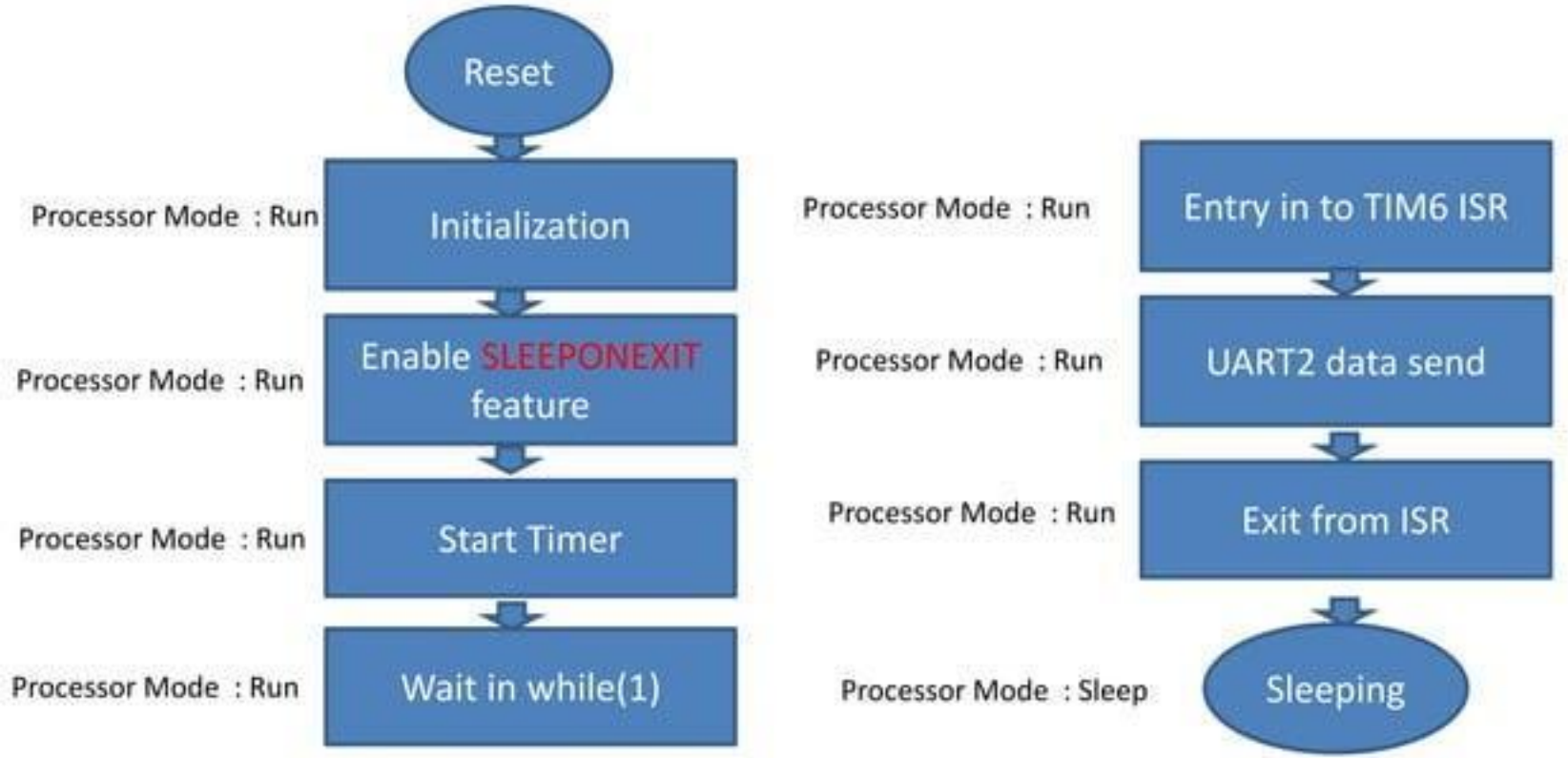
Entering SLEEP mode using Sleep-on-Exit feature

Note :

In interrupt-driven applications, do not enable Sleep-On-Exit feature too early during the initialization. Otherwise if the processor receives an interrupt request during the initialization process, it will enter sleep automatically after the interrupt handler executed before the rest of the initialization process completes.

Exercise

- Write an application in which TIM6 triggers Update interrupt for every 10ms and in the ISR of TIM6 send some data over UART2.
- Measure the Current consumption without sleep mode
- Measure the Current Consumption With Sleep mode (*Enter sleep mode using SLEEPONEXIT feature*)

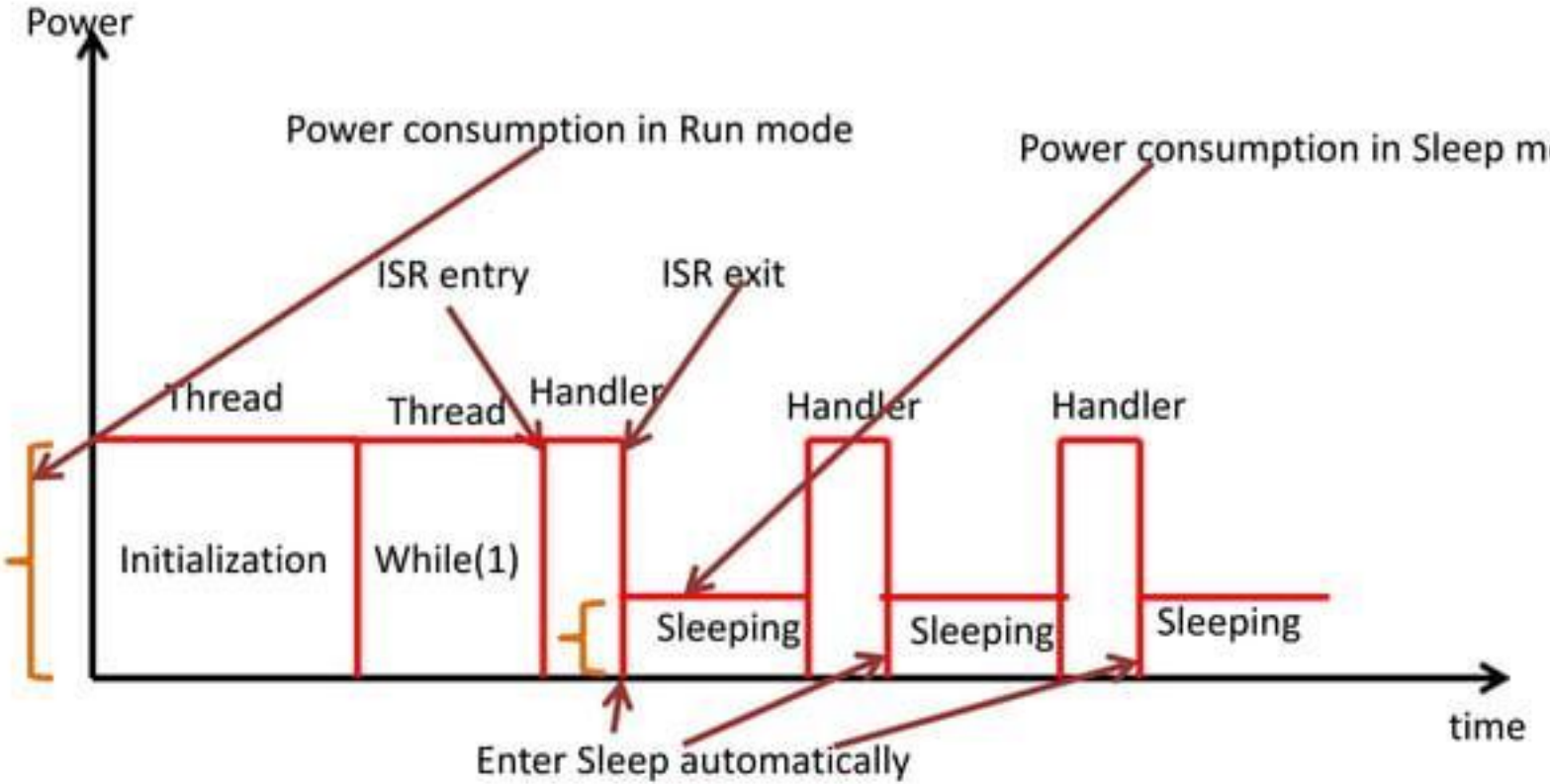


Waking up from SLEEPONEXIT

- when the processor enters sleep mode using the Sleep-On-Exit feature or executing the WFI instruction the processor stops instruction execution, enters sleep mode and wakes up when a higher priority interrupt request arrives and needs to be serviced.(This what ARM Says, but MCU vendor may implement wakeup procedure differently , e.g in ST's Case processor wakes up for any interrupt, i.e priority don't care)

- If the processor enters sleep in an exception handler, and if the newly arrived interrupt request has the same or lower priority compared to the current exception, the processor will not wake up and interrupt will remain in pending state. (This what ARM Says, but MCU vendor may implement wakeup procedure differently)

- The processor can also be woken up by a halt request from the debugger or by reset



WFI(Wait For Interrupt)

- It is a 16 bit Thumb instruction
- When Cortex-Mx (0, 0+,3,4,7…) processor executes a WFI instruction it stops executing instructions and enters sleep mode(Clocks to the processor is stopped), until the arrival of an interrupt or if the processor enters a debug state
- Enter sleep unconditionally
- Can be used with both normal sleep and deep sleep mode

WFI(Wait For Interrupt)

- The WFI instruction is a Cortex-Mx instruction which cannot be directly accessible by ANSI C. The CMSIS (*Cortex Microcontroller Software Interface Standard*) provides an intrinsic function to generate a WFI instruction and is supported by C compiler. If a C compiler does not support the WFI intrinsic function, then the user will have to use assembly code to execute WFI instruction.

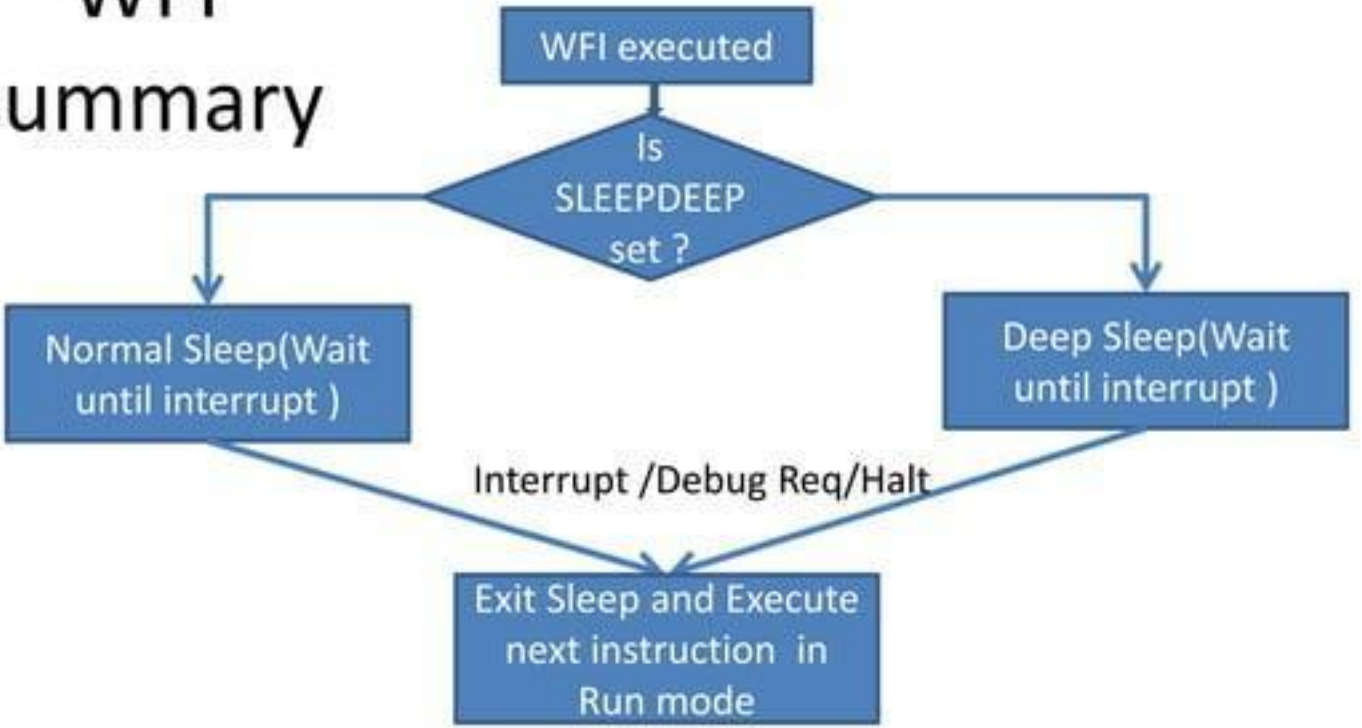
Waking up from WFI

- When processor goes to sleep executing WFI instruction in a thread mode , any interrupt request , debug request or reset can wake up the processor.

- If the processor executes WFI instruction in an exception handler and enters sleep mode then only higher priority interrupt request (> current level) can only wake up the processor. If the newly arrived interrupt request has the same or lower priority compared to the current exception, the processor will not wake up and the newly issued interrupt will remain in pending state. .(This what ARM Says, but MCU vendor may implement wakeup procedure differently)

- The processor can also be woken up by a halt request from the debugger or by reset

WFI summary



Exercise

- Write an application in which TIM6 triggers Update interrupt for every 10ms and in the ISR of TIM6 send some data over UART2.
- Measure the Current consumption without sleep mode
- Measure the Current Consumption With Sleep mode (*Enter sleep mode using WFI Instruction*)

Tips to Reduce the Power Consumption

- Stopping clock to the processor and to some or all the peripherals (clock gating)
- Reducing Clock frequency.
- Reducing Voltage to various parts of the microcontroller
- Turning off the power supply to some parts of the microcontroller
- Keeping unused pins in the analog mode

- **500** milliamps for one hour before being "dead”
- 500 1 hour
- 2.7 ? 185 hour – 1 week

WFE(Wait for Event)

- It is a 16 bit Thumb instruction
- Enter sleep conditionally
- Its intended usage is as part of a power saving strategy in spinlock loops in RTOS
- It is possible that execution of a WFE instruction will complete immediately without causing the processor to go into a low power (sleeping) state
- WFE success depends upon event register of the processor

Event register of the Processor

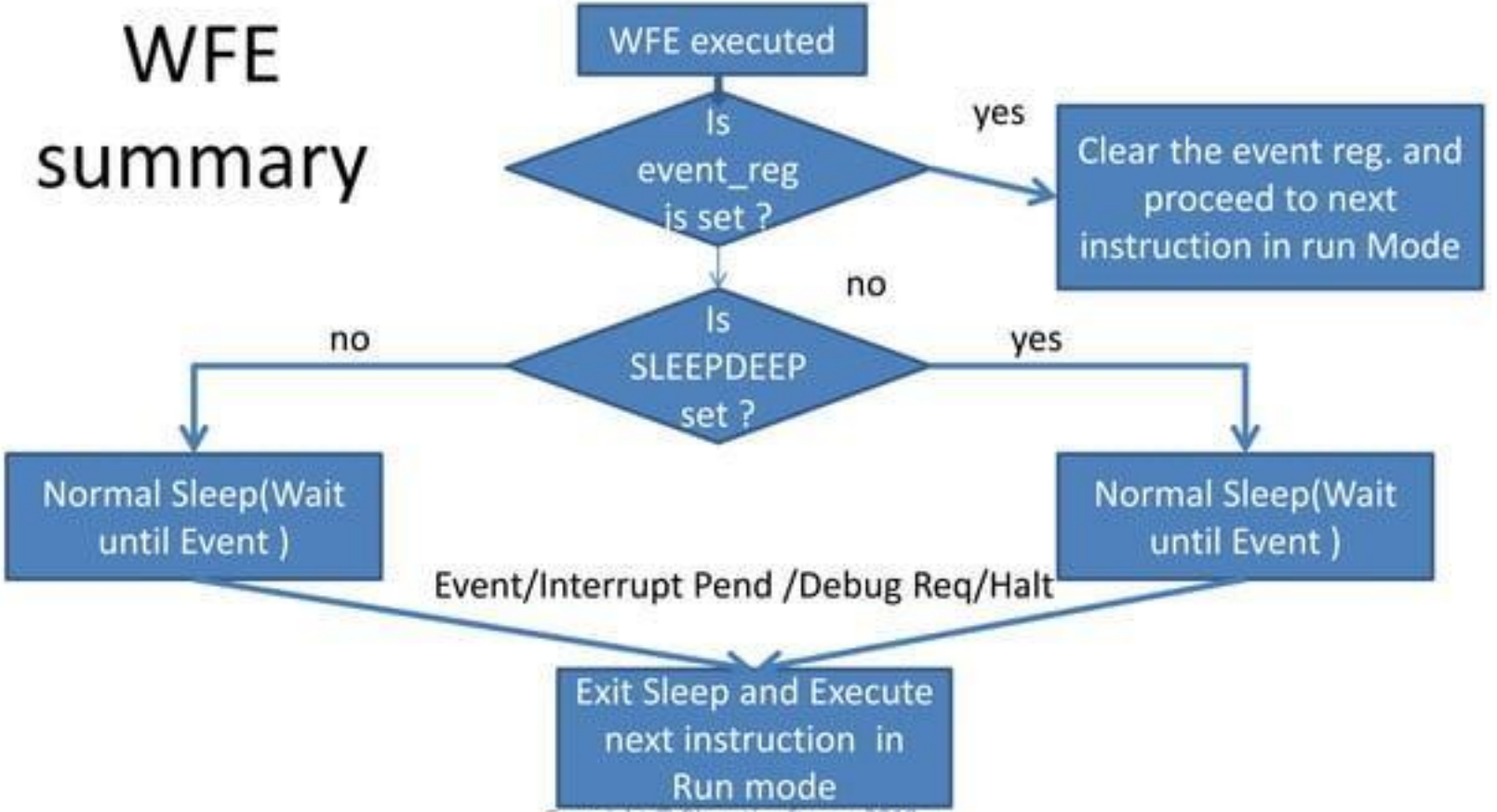
- Inside a Cortex-M processor, there is a single-bit event register.
- This event register bit will be 0, after reset
- The event register is used to hold an event which happened in the past,
- When set, an Event Register indicates that an event has occurred

- When any event occurs this bit is set to 1
- When you execute WFE and if WFE sees that event register bit is 1, it makes it 0 and doesn't cause processor to go to sleep. (*A WFE instruction clears the Event Register*.)
- WFE succeeds (i.e puts processor to sleep) only when event register bit is 0 at the time of executing the WFE instruction .
- Software cannot read or write the value of the Event Register directly.

Event register of the Processor

- The event register can be set by any of the following events:
  - An interrupt request arrives and need servicing
  - Exception entrance and exception exit
  - New pending interrupts (only when SEVONPEND bit in SCR is set), even if the interrupts are disabled
  - An external event signal from on-chip hardware (MCU specific)
  - Execution of an SEV (Send Event) instruction
  - Debug event
- When multiple events occur while the processor is awake, they will be treated as just one event because the event register is only one bit.

WFE summary



Wakeup from WFE

- When the WFE instruction is used to enter sleep, it can be woken up by
  - The execution of an SEV(Send Event) instruction
  - Any exception entering the Pending state if SEVONPEND in the System Control Register is set
  - An asynchronous exception at a priority that preempts any currently active exceptions
  - An event from another processor/peripheral

- WFE can also be woken up by interrupt requests if they have a higher priority than the current interrupt's priority level,

- The SEVONPEND feature can wake up the processor from WFE sleep even if the priority level of the newly pended interrupt is at the same or lower level than the current interrupt. However, in this case, the processor will not execute the interrupt handler and will resume program execution from the instruction following the WFE.

WFI and WFE sleep wake-up behavior

WFI behavior	PRIMASK	SEVONPEND	Wake Up	ISR execution
IRQ priority > current level	0	N/A	Yes	Yes
IRQ priority <= current level	0	N/A	No	No
IRQ priority > current level	1	N/A	Yes	No
IRQ priority <= current level	1	N/A	No	No

WFI and WFE sleep wake-up behavior

WFE behavior	PRIMASK	SEVONPEND	Wake Up	ISR execution
IRQ priority > current level	0	0	Yes	Yes
IRQ priority <= current level	0	0	No	No
IRQ priority > current level	0	1	Yes	Yes
IRQ priority <= current level(or IRQ disabled	0	1	Yes	No
IRQ priority > current level	1	0	No	No
IRQ priority <= current level	1	0	No	No
IRQ priority > current level	1	1	Yes	No
IRQ priority <= current level	1	1	Yes	No

Exercise

- Write an application in which TIM6 triggers Update interrupt for every 10ms and in the ISR of TIM6 send some data over UART2.
- Measure the Current consumption without sleep mode
- Measure the Current Consumption With Sleep mode (*Enter sleep mode using WFE Instruction*)

Differences and similarities of WFI and WFE

Similarities

- Wake up on Interrupt requests
- Wake up by debug events
- Can be used to produce normal or deep sleep

Differences

- WFI puts the processor immediately to sleep, where as WFE puts the processor to sleep only if event register value is 0
- No relation between event register and WFI , but WFE works along with event register .
- New pending of a disabled interrupt can wake up the processor from WFE sleep if SEVONPEND is set
- WFE can be woken up by an external event signal

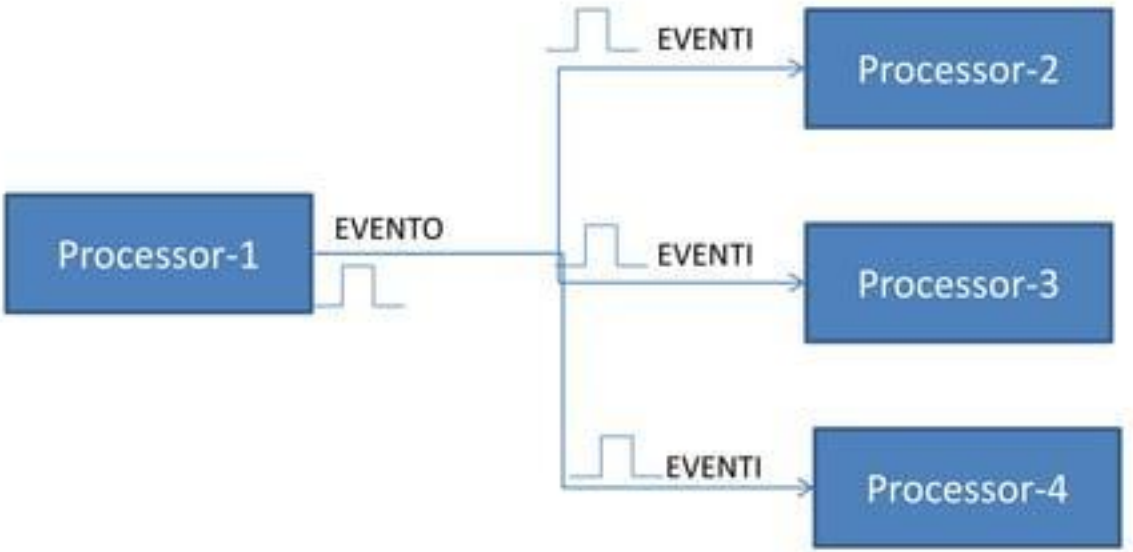
When to use what ?

Use WFI instruction in interrupt driven applications and stand Alone application.

Use WFE in idle loops , Spinlocks , busy waiting and RTOS related scenarios where processor will go to sleep if there are no events are pending to be processed.

WFI is targeted at entering either standby, dormant or shutdown mode, where an interrupt is required to wake-up the processor.

usage for WFE



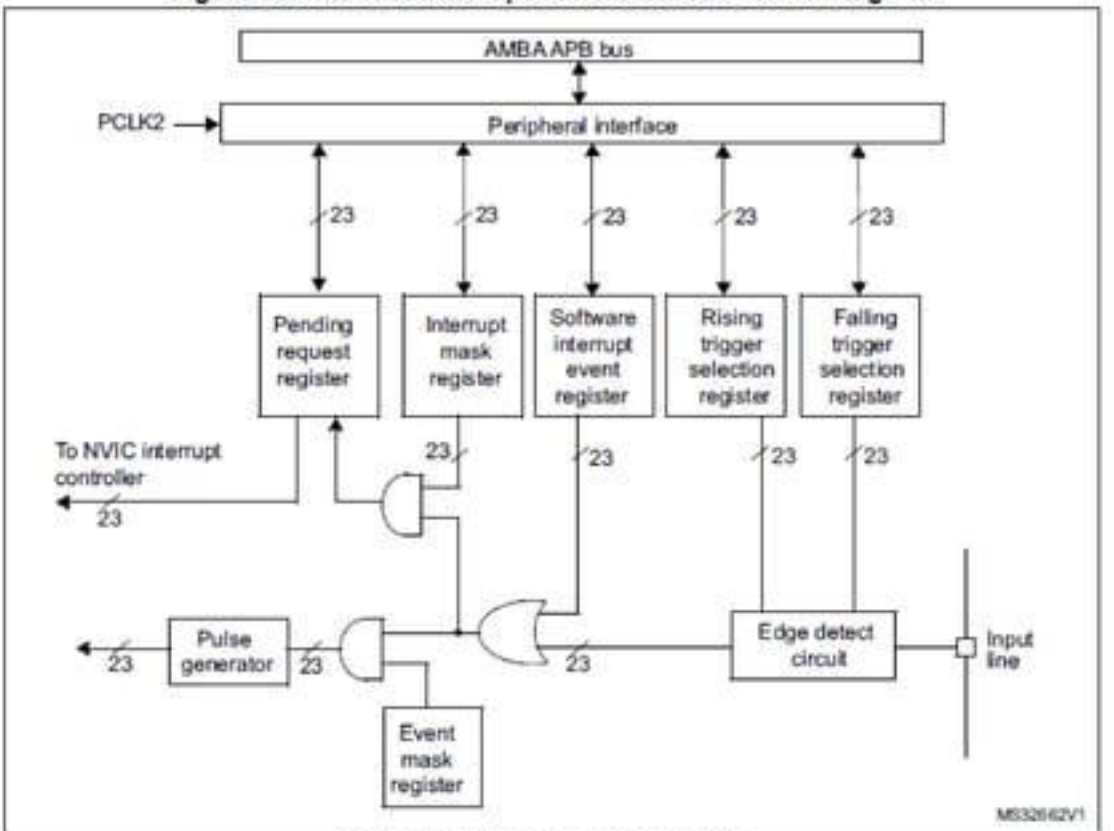
How to generate events ?

- The STM32F446xx microcontrollers are able to handle external or internal events in order to wake up the core (WFE). The wakeup event can be generated by:
  - enabling an interrupt in the peripheral control register but not in the NVIC, and enabling
- the SEVONPEND bit in the Cortex®-M4 with FPU System Control register. When the
- MCU resumes from WFE, the peripheral interrupt pending bit and the peripheral NVIC
- IRQ channel pending bit (in the NVIC interrupt clear pending register) have to be
- cleared.
- • or configuring an external or internal EXTI line in event mode. When the CPU resumes
- from WFE, it is not necessary to clear the peripheral interrupt pending bit or the NVIC
- IRQ channel pending bit as the pending bit corresponding to the event line is not set.

How to generate events in STM32 MCU

1. Execute the SEV instruction .
2. Use Peripheral Interrupt as an event .
   - Make SEVONPEND bit as 1
   - Disable peripheral interrupt in the NVIC
   - Make peripheral issue an interrupt
   - Applicable to all peripherals
3. Use peripheral event
   - Applicable to Some of the peripherals

Figure 29. External interrupt/event controller block diagram



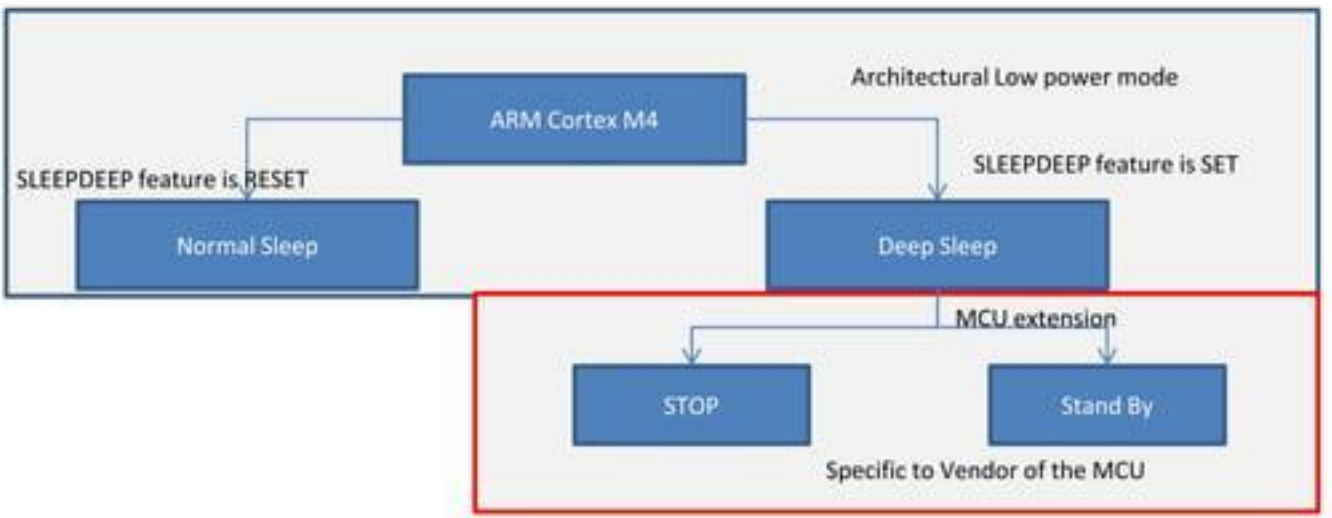
How to generate events ?

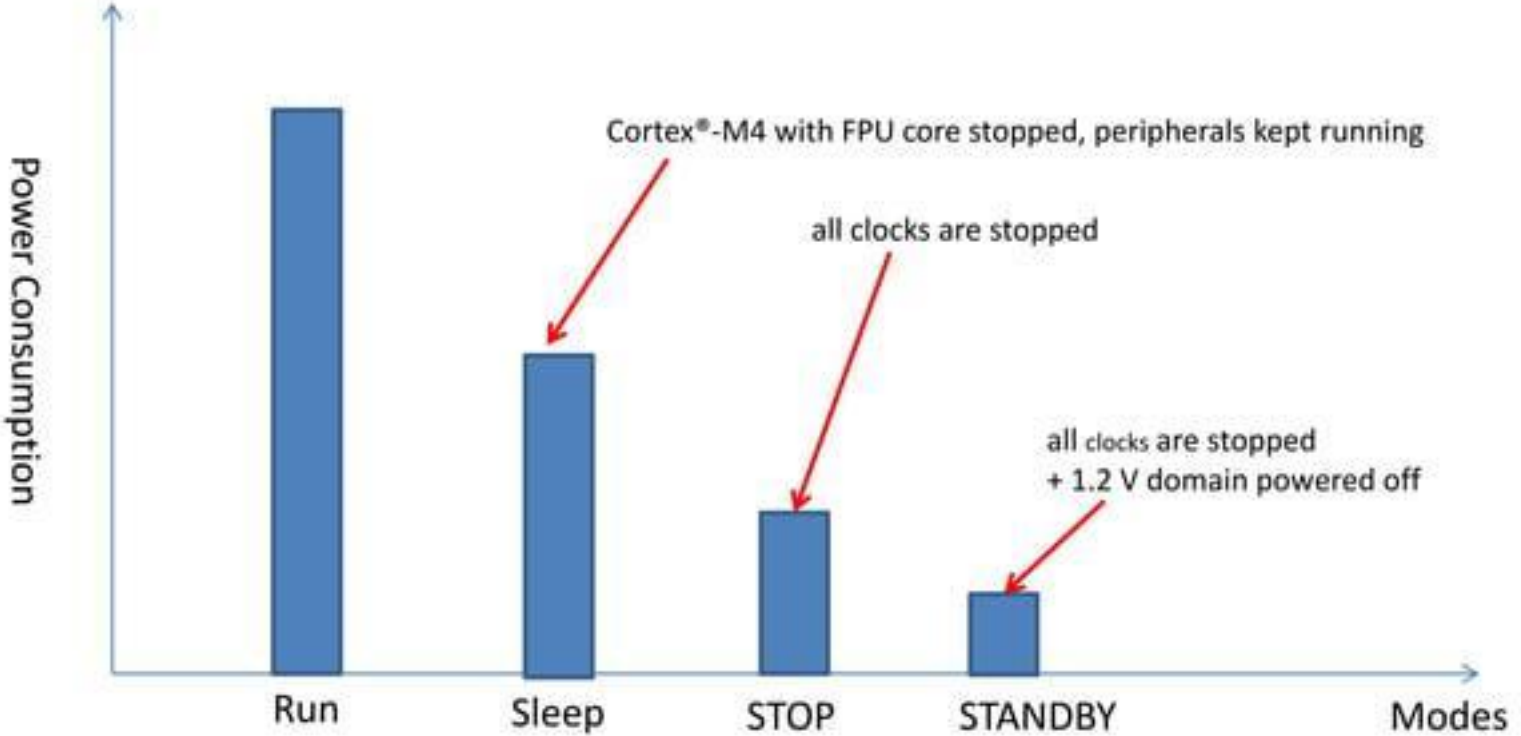
- To generate the event, the event line should be configured and enabled
- This is done by programming the two trigger registers with the desired edge detection and by enabling the event request by writing a '1' to the corresponding bit in the event mask register
- When the
- selected edge occurs on the event line, an event pulse is generated
- pending bit
- corresponding to the event line is not set.
- Configure the mask bits of the 23 event lines (EXTI_EMR)
- Configure the Trigger selection bits of the event lines (EXTI_RTSR and EXTI_FTSR)

Device Specific (MCU) Low Power Modes

Device Specific (MCU) Low Power Modes

Low Power modes of the MCU is further extended by the device vendor using Power modes given by arm cortex M Processor

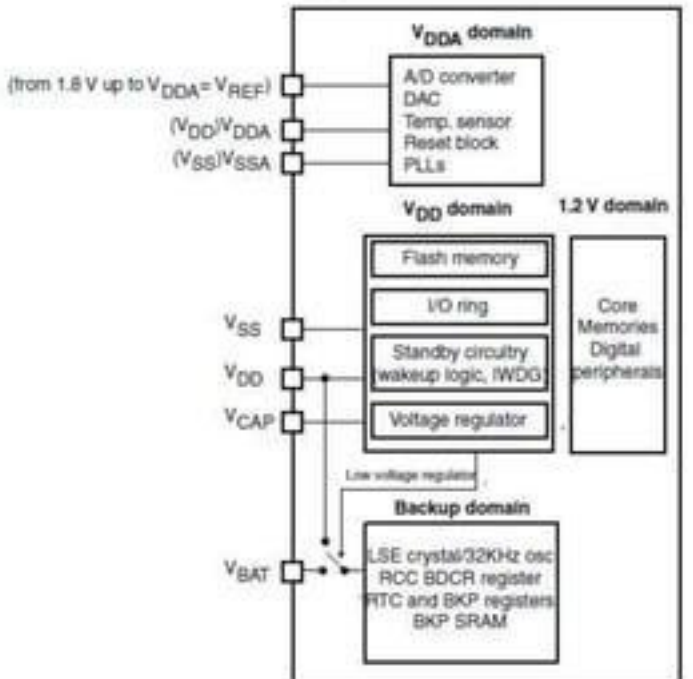




Device Specific (MCU) Low Power Modes

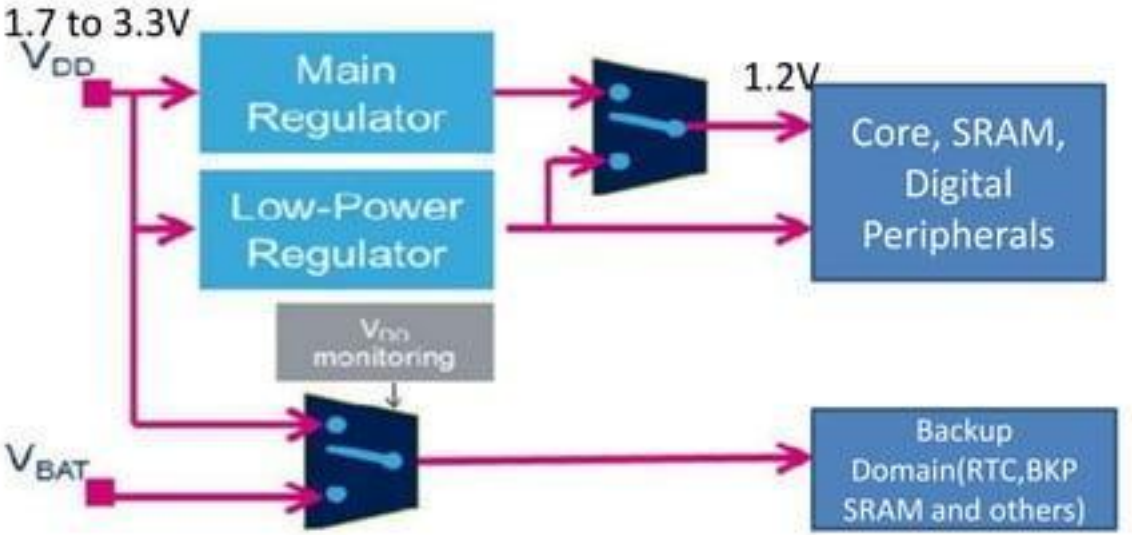
- **Sleep mode**
  - In Sleep mode, only the CPU is stopped
  - All peripherals continue to operate and can wake up the CPU when an interrupt/event occurs
  - You can always do additional settings to save some more power (like running the CPU at lower clock , disabling the peripheral clocks , etc)

Voltage Domains



VBAT

STM32F4x Voltage Regulator

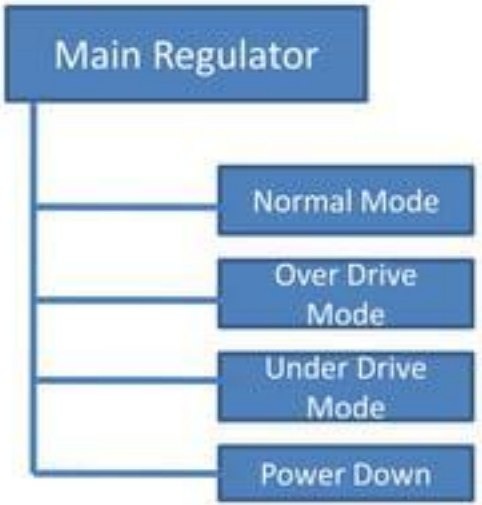


An embedded linear voltage regulator supplies all the digital circuitries except for the backup domain and the Standby circuitry. The regulator output voltage is around 1.2 V.

Regulator modes

You can select either MR or LPR by configuring the LPDS bit in the **PWR power control register (PWR_CR)**

Main regulator Modes



Normal mode: The CPU and core logic operate at maximum frequency at a given voltage scaling (scale 1, scale 2 or scale 3) Voltage scaling can be configured by VOS bit in PWR_CR

Use “Normal Mode” in your RUN mode or in STOP Mode

Voltage Scaling

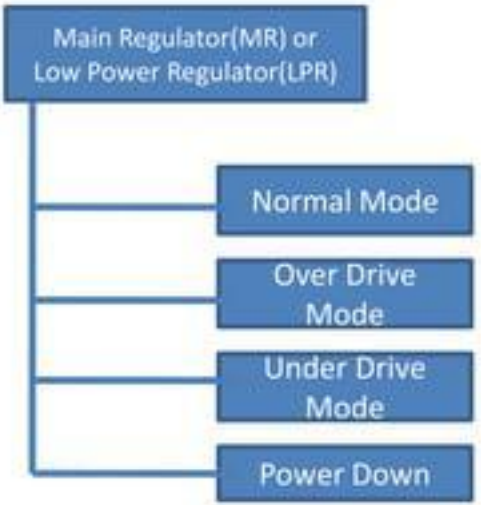
- There are 3 Voltage Scales are available in STM32F4x Microcontroller (Scale 1 , scale 2, scale 3). The main goal of this is to reduce the power consumption by controlling the output voltage of the voltage regulator with respect to operating frequency
- *Refer* ***Table 16. General operating conditions In Datasheet***

Voltage Scaling

The voltage scaling and over-drive mode are adjusted to fHCLK frequency as follows:

- ❑Scale 3 for fHCLK ≤ 120 MHz
- ❑Scale 2 for 120 MHz < fHCLK ≤ 144 MHz
- ❑Scale 1 for 144 MHz < fHCLK ≤ 180 MHz. The over-drive is only ON at 180 MHz.

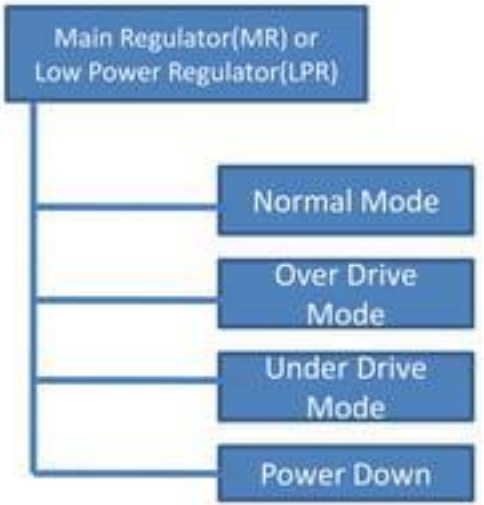
Main regulator & Low power regulator Modes



Over Drive mode is used to run the CPU and Core logic at maximum possible frequency . In STM32F446RE microcontroller it is 180Mhz. The output voltage of the Main regulator will be more than typical 1.2V thus consuming more power of your application

If your application demands using 180Mz, then you have to turn on the over drive mode.

Main regulator Modes



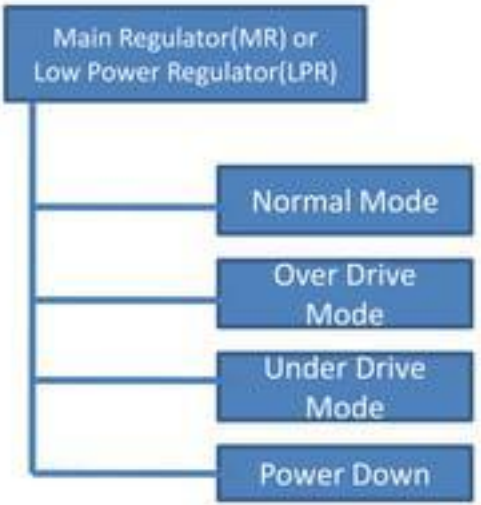
Under-drive mode: the 1.2 V domain is preserved in reduced leakage mode

This mode of the regulator is can be used in STOP mode.

If the Under-drive mode was enabled, it is automatically disabled after

* exiting Stop mode.
* When the voltage regulator operates in Under-drive mode, an additional
* startup delay is induced when waking up from Stop mode.

Main regulator Modes



Power-down is used in Standby mode. The Power-down mode is activated only when entering in Standby mode. The regulator output is in high impedance inducing zero consumption. The contents of the registers and SRAM are lost

Summary

Table 14. Voltage regulator configuration mode versus device operating mode(1)

Voltage regulator configuration	Run mode	Sleep mode	Stop mode	Standby mode
Normal mode	MR	MR	MR or LPR	-
Low-voltage mode	-	-	MR or LPR	-
Over-drive mode(2)	MR	MR	-	-
Under-drive mode	-	-	MR or LPR	-
Power-down mode	-	-	-	Yes

1. '-' means that the corresponding configuration is not available.
2. The over-drive mode is not available when V_{DD} = 1.8 to 2.1 V.

Device Specific (MCU) Low Power Modes

- **STOP mode**
  - The Stop mode achieves the lower power consumption compared to sleep mode.
  - All clocks in the 1.2 V domain are stopped (clocks to CPU, SRAM, digital peripherals)
  - Since Voltage regulator is not powered down , SRAM content will be retained
  - You can either use MR or LPR
  - The voltage regulator can be in either Normal mode or under drive mode
  - PLL, the HSI RC and the HSE crystal oscillators are disabled.
  - The device can be woken up from the Stop mode by any of the EXTI line (the EXTI line source can be one of the 16 external lines, the PVD output, the RTC alarm / wakeup / tamper / time stamp events, the USB OTG FS/HS wakeup).

Device Specific (MCU) Low Power Modes

- **Stand by mode**
  - The Standby mode is used to achieve the lowest power consumption
  - The internal voltage regulator is powered down so that the entire 1.2 V domain is powered off
  - SRAM loses its data since voltage regulator is powered down
  - The PLL, the HSI RC and the HSE crystal oscillators are also switched off
  - After entering Standby mode, the SRAM and register contents are lost except for registers in the backup domain and the backup SRAM when selected.
  - The device exits the Standby mode when an external reset (NRST pin), an IWDG reset, a rising edge on the WKUP pin, or an RTC alarm / wakeup / tamper /time stamp event occurs.

STOP Mode wakeup time

- In stop mode, wakeup time depends on the voltage regulator and the Flash memory wakeup time
- For example, if the voltage regulator is in run mode and the MCU is configured in stop mode, the wakeup time is about 13 µs.
- However when the voltage regulator and the Flash
- memory are configured in low power mode, the wakeup time increases to reach 110 µs(for
- more details about the wakeup time, please refer to the STM32F2xx datasheet).

STOP Mode wakeup time

- Observation:
- ● Current consumption and wakeup time in stop mode depends on the voltage regulator
- configuration (run mode or low power mode) and Flash memory configuration (stop
- mode or deep power mode)
- ● A lower current consumption means a longer the wakeup time

Standby mode

- Standby mode is based on the Cortex-Mx deep sleep mode, with the voltage regulator
- disabled. The 1.2 V domain is consequently powered off. The PLL, the HSI oscillator and
- the HSE oscillator are also switched off. SRAM and register contents are lost except for
- registers in the Backup domain and Standby circuitry.
- The Standby mode current consumption depends on the backup SRAM and RTC
- configuration, which is between 5.8 µA when both backup SRAM and RTC are enabled, and
- 2.5 µA when both backup SRAM and RTC are disabled

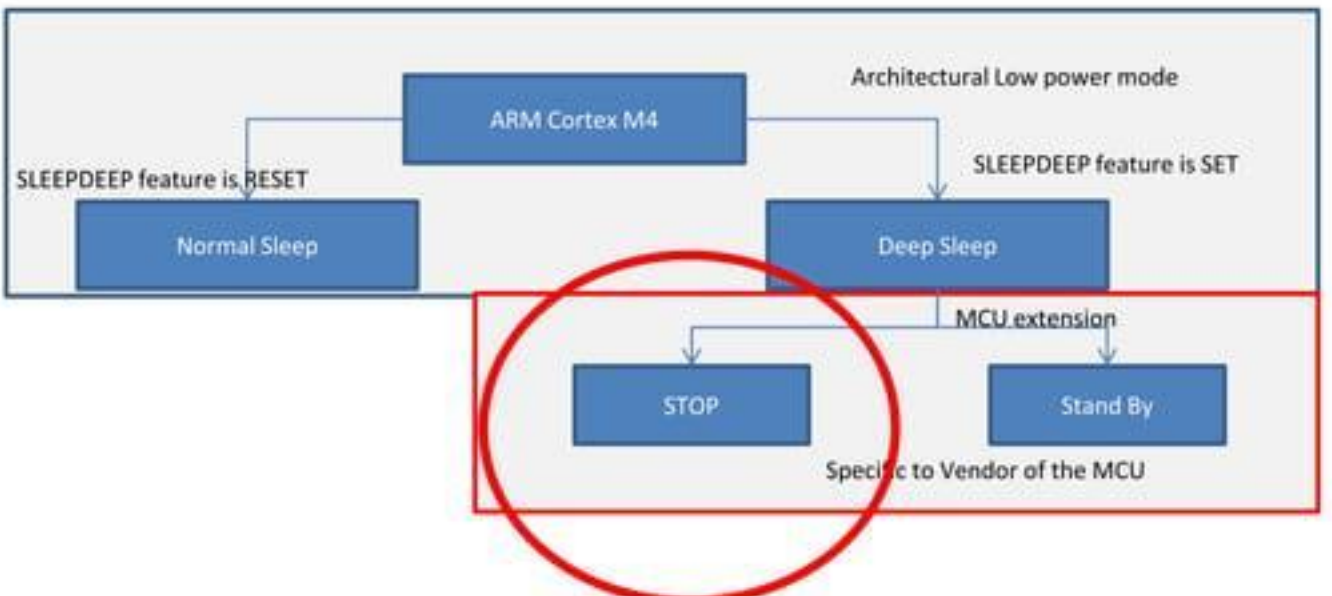
Stand by Mode wakeup time

- Observation:
- ● Standby mode returns the lowest power consumption
- ● The wakeup time from Standby mode is the highest wakeup time. It is about 375 µs
- (refer to the section “Wakeup time from low-power mode” in the STM32F20xx/21xx
- datasheet)

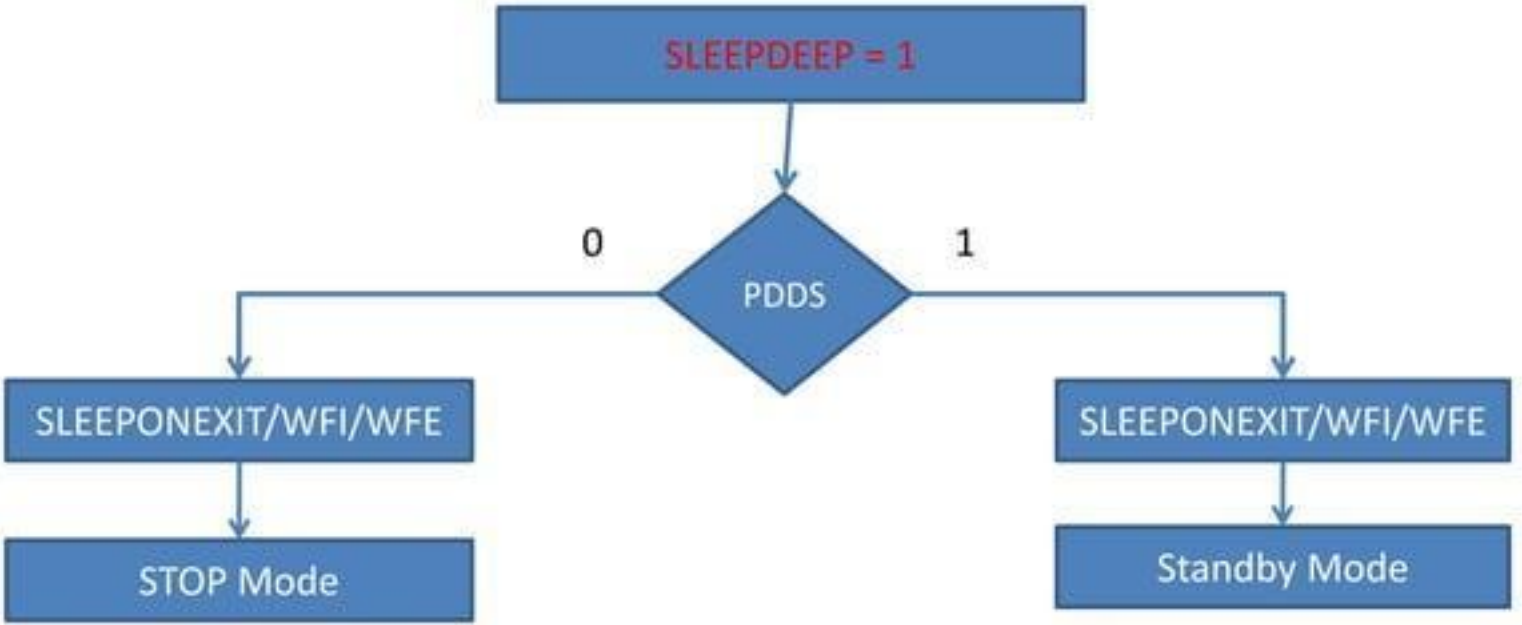
Conclusion

- Conclusion:
- The global wakeup time of the device can be viewed as the sum of the following:
- ● Wakeup time of the main voltage regulator (if it is switched off during the low-power
- mode)
- ● Stabilization time of the oscillator (if it is switched off during the low-power mode)
- ● Wakeup time of the Flash memory (if it is switched off during the low-power mode)
- ● Interrupt latency for the wake up event trigger
- The results above shows the trade-off between consumption and wakeup time in the
- STM32F2. Generally, the lower the power consumption is, the longer the wakeup time. You
- should therefore try to find the best trade-off according to the application constraints.

How to put MCU in STOP mode?

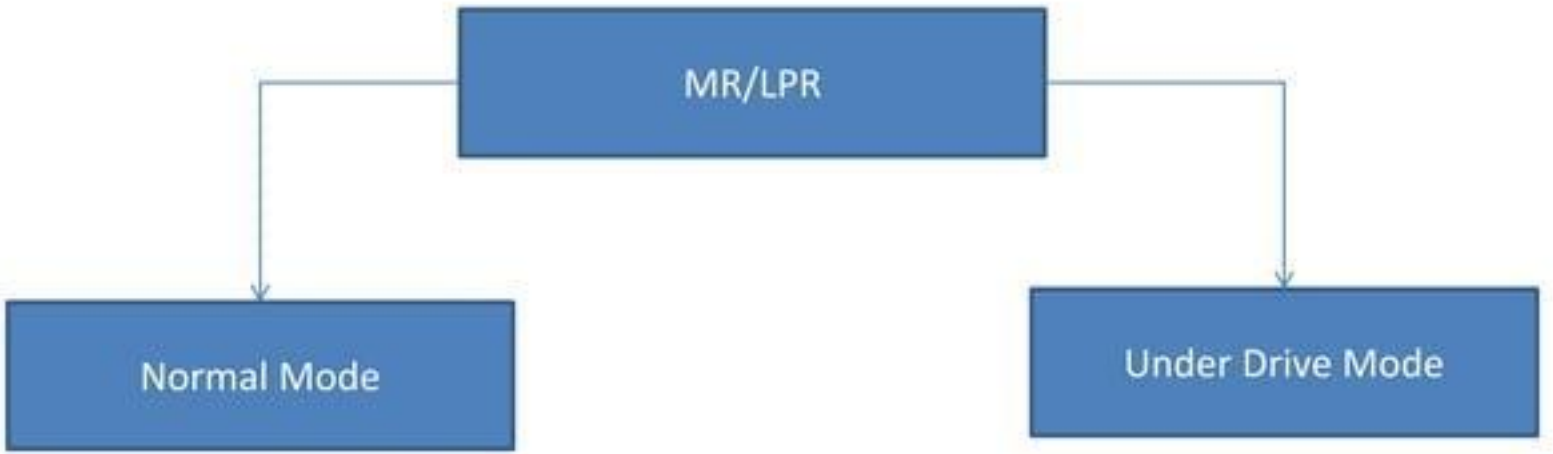


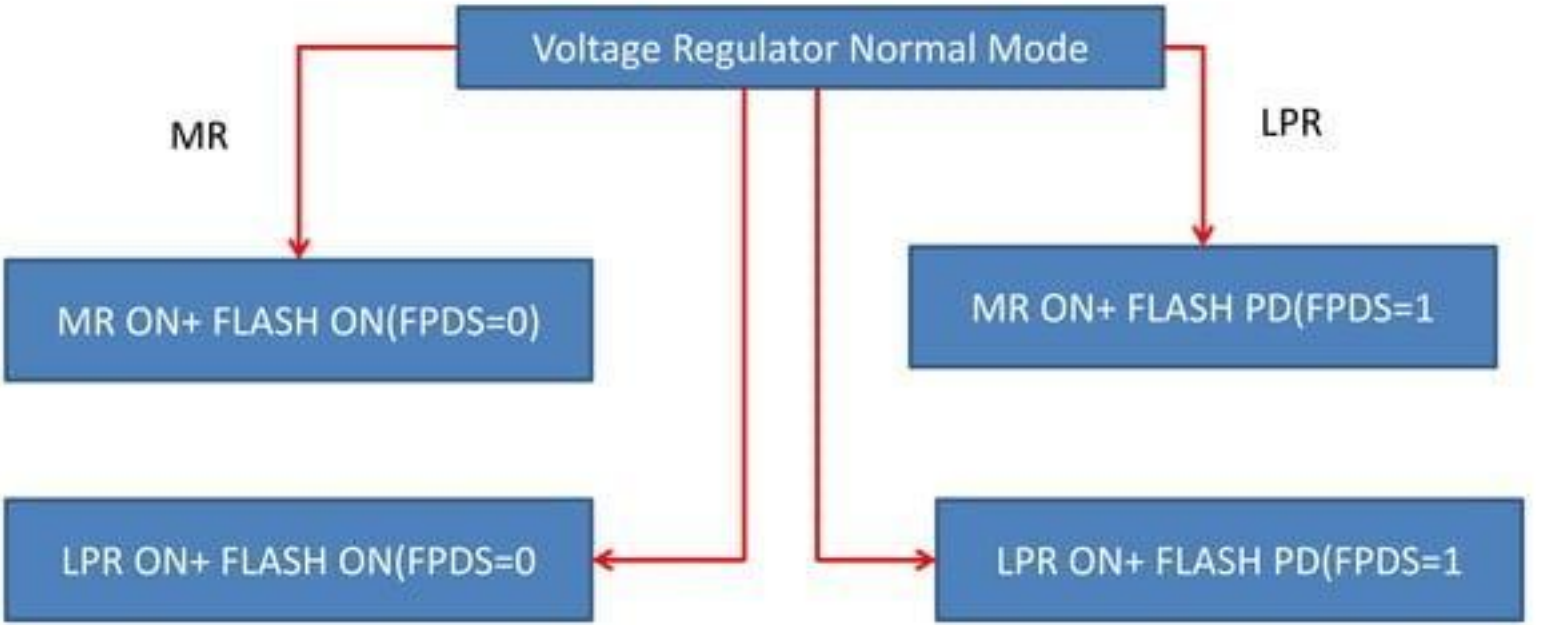
How to put MCU in STOP mode?

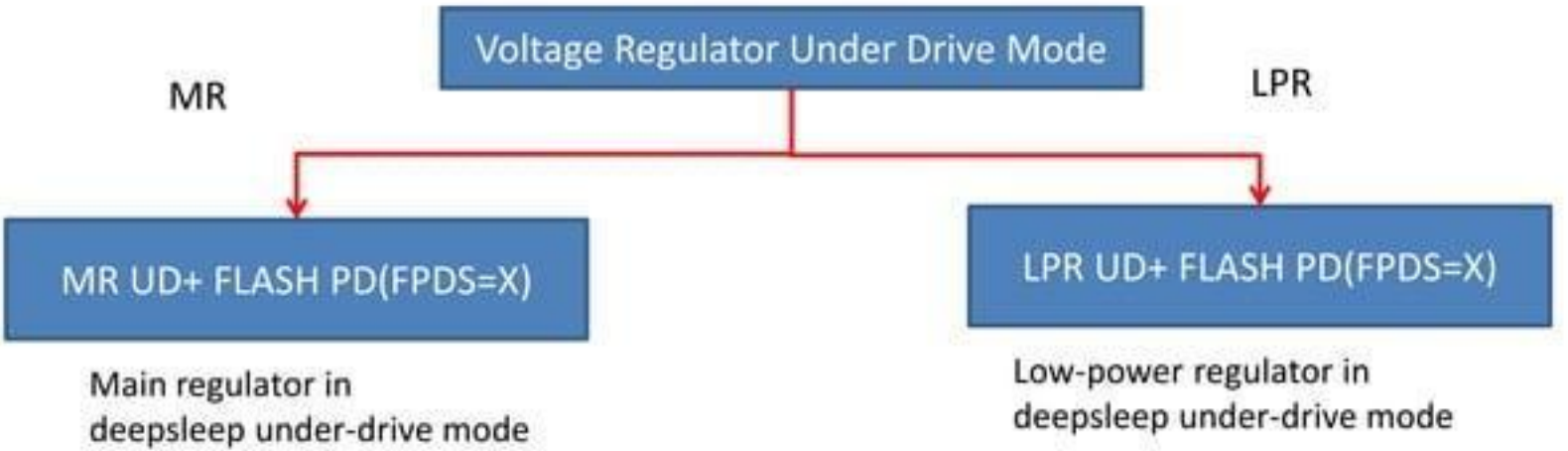


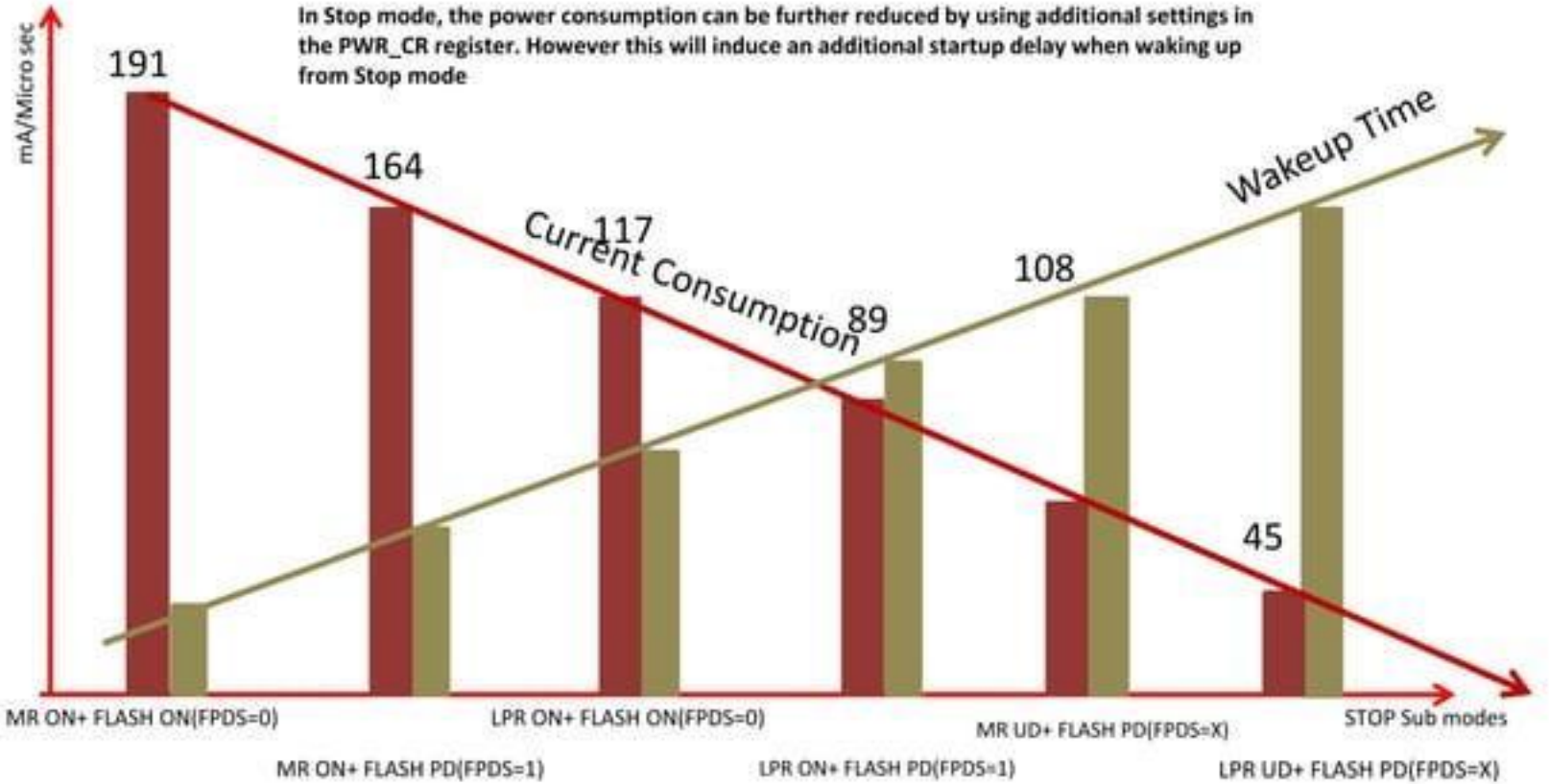
Voltage Regulator Settings in STOP Mode

Remember Voltage Regulator will not be OFF in STOP mode









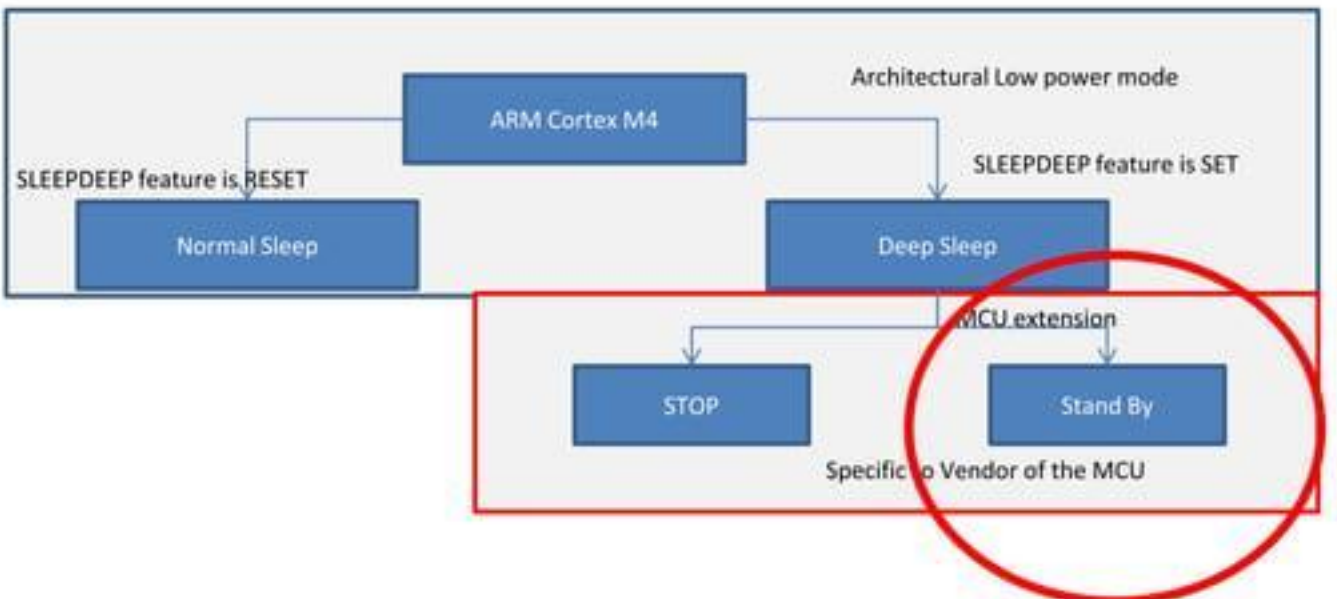
Exercise

- Test the Current Consumption in below STOP mode conditions
  - STOP + MR ON + FLASH ON
  - STOP + MR ON + FLASH Power Down
  - STOP+ LPR ON + FLASH ON
  - STOP + LPR ON + FLASH Power Down
  - STOP + MR Under Drive + FLASH Power Down
  - STOP + LPR Under Driver + FLASH Power Down
  - Waking up using Button interrupt (EXTI)

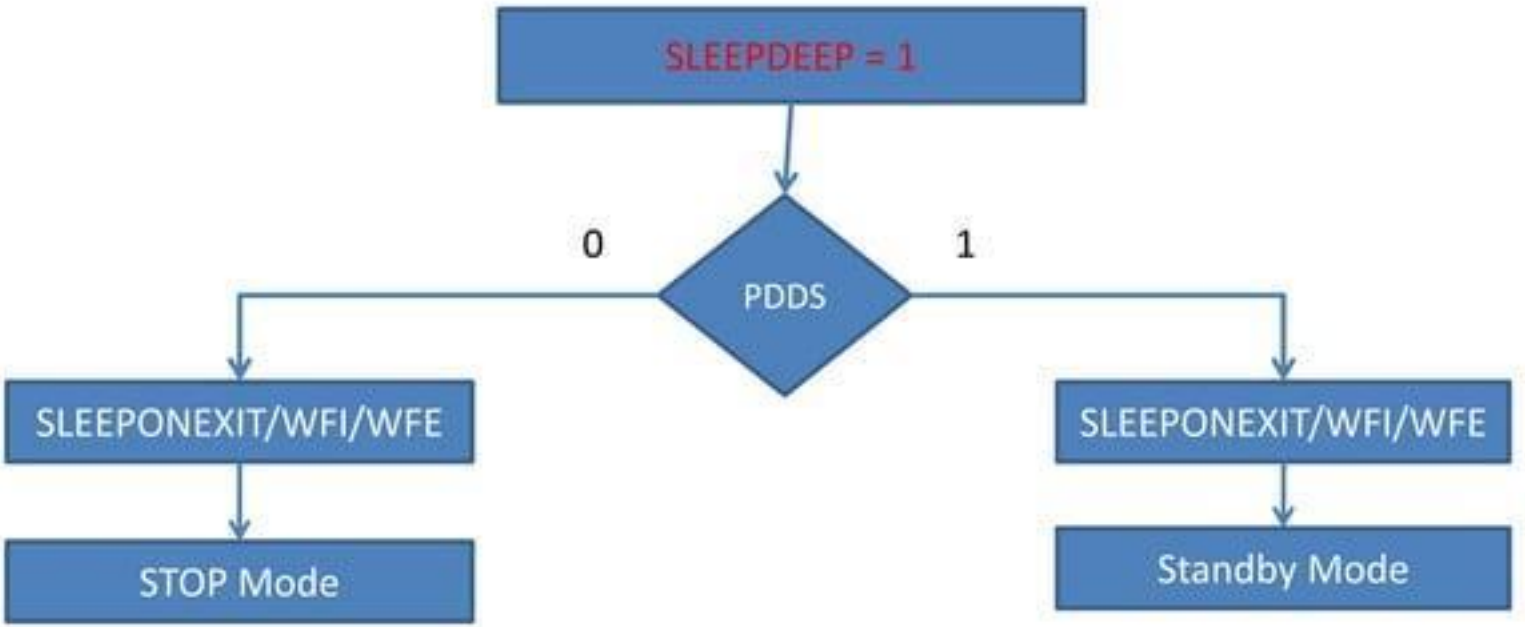
Wakeup From STOP Mode

- When exiting Stop mode by issuing an interrupt or a wakeup event, the HSI RC oscillator is selected as system clock
- If the Under-drive mode was enabled, it is automatically disabled after exiting Stop mode.
- When the voltage regulator operates in low-power or low voltage mode, an additional startup delay is incurred when waking up from Stop mode. By keeping the internal regulator ON during Stop mode, the consumption is higher although the startup time is reduced.
- When the voltage regulator operates in Under-drive mode, an additional startup delay is induced when waking up from Stop mode.
- **Wakeup latency Refer to *Table 17: Stop operating modes in RM***

How to put MCU in Standby Mode?



How to put MCU in Standby Mode?



Who is alive in Standby ?

- RTC registers and RTC backup registers
- backup SRAM
- Standby circuitry(WKUP Logic and IWDG)
- LSI and LSE

secondary clock sources:

• 32 kHz low-speed internal RC (LSI RC) which drives the independent watchdog and, optionally, the RTC used for Auto-wakeup from the Stop/Standby mode.

• 32.768 kHz low-speed external crystal (LSE crystal) which optionally drives the RTC clock (RTCCLK)

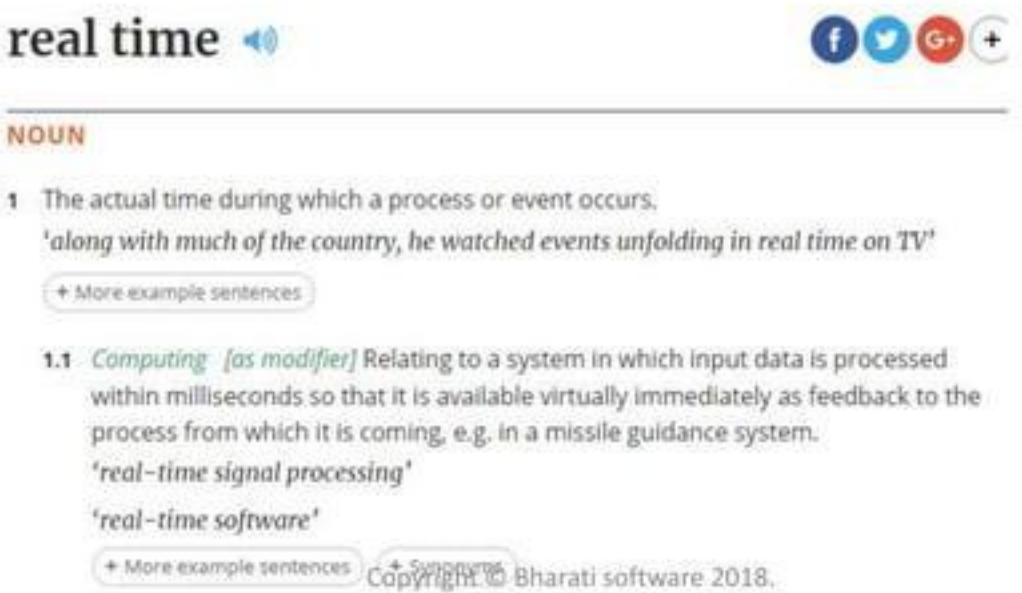
Each clock source can be switched on or off independently when it is not used, to optimize power consumption.

Exercise

- Test the Current Consumption in below STANDBY mode conditions
  - STANDBY + BKP SRAM ON + RTC ON
    (wakeup using RTC automatic wake up after 20s)
  - STANDBY + BKP SRAM OFF + RTC ON ON
    (wakeup using RTC automatic wake up after 20s)
  - STANDBY + BKP SRAM ON + RTC OFF
    (wakeup using wakeup pin)
  - STANDBY + BKP SRAM OFF + RTC OFF
    (wakeup using wakeup pin)

RTC Intro

- What is “Real time” ??



real time

NOUN

1 The actual time during which a process or event occurs.
'along with much of the country, he watched events unfolding in real time on TV'

+ More example sentences

1.1 *Computing* *[as modifier]* Relating to a system in which input data is processed within milliseconds so that it is available virtually immediately as feedback to the process from which it is coming, e.g. in a missile guidance system.
'real-time signal processing'
'real-time software'

+ More example sentences + Synonyms

RTC Intro

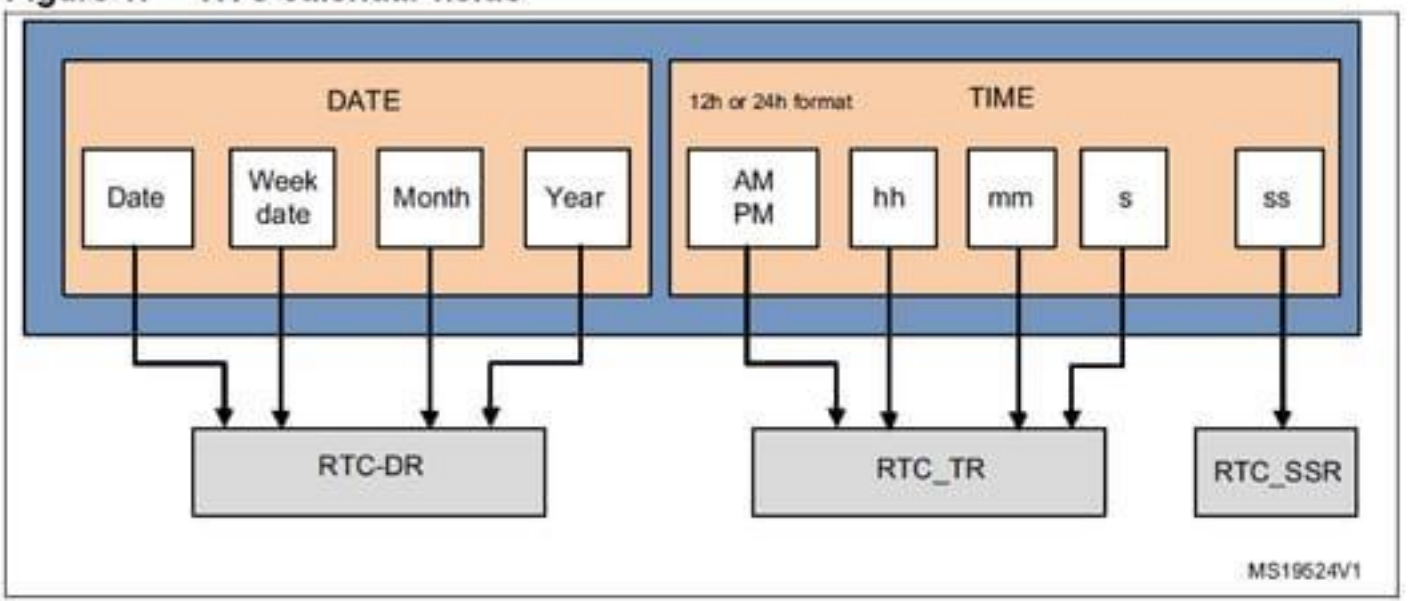
- A real-time clock (RTC) is a peripheral of the MCU that keeps track of the current time and date information
- The real-time clock (RTC) embedded in STM32 microcontrollers acts as an independent BCD timer
- Although RTCs are often used in personal computers, servers and embedded systems, they are also present in almost any electronic device that requires accurate time keeping.
- As long as the supply voltage remains in the operating range, the RTC never stops regardless of the device status (Run mode, low power mode or under reset).

RTC Main features

- Calendar with sub-seconds, seconds, minutes, hours (12h or 24h format), day (day of week), date (day of month), month, and year.
- Two programmable alarms with interrupt function. The alarms can be triggered by any combination of the calendar fields.
- Automatic wakeup unit generating a periodic flag that triggers an automatic wakeup interrupt.
- 20 backup registers (80 bytes). The backup registers are reset when a tamper detection event occurs

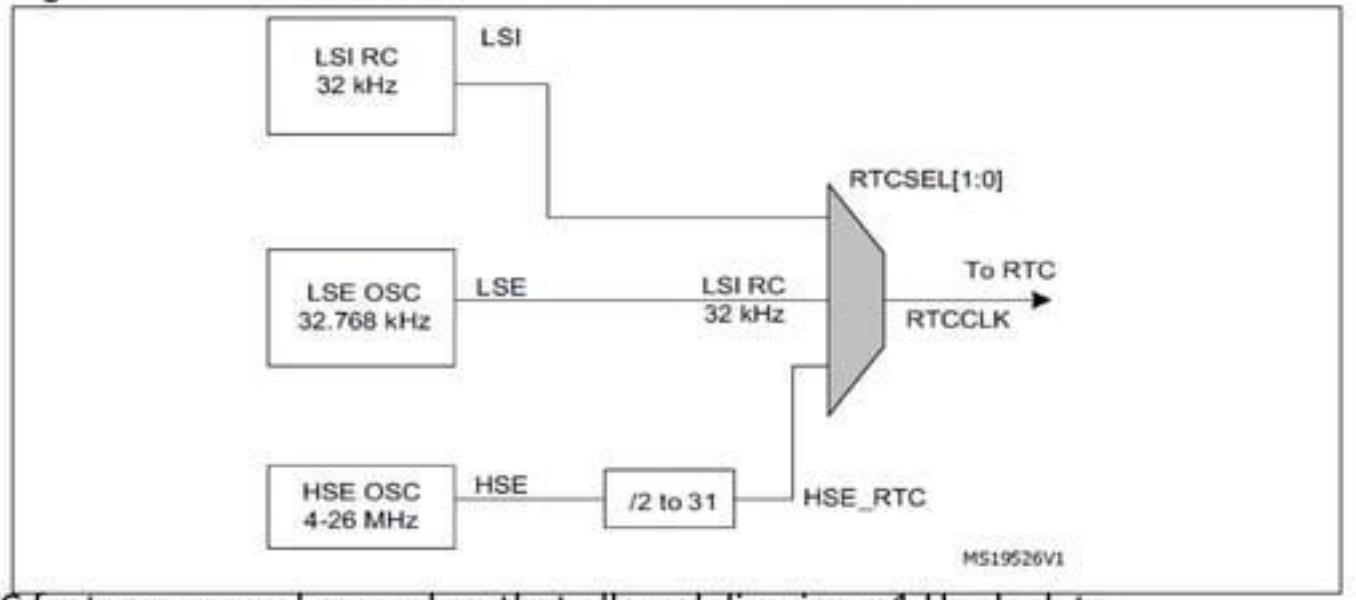
RTC calendar

Figure 1. RTC calendar fields



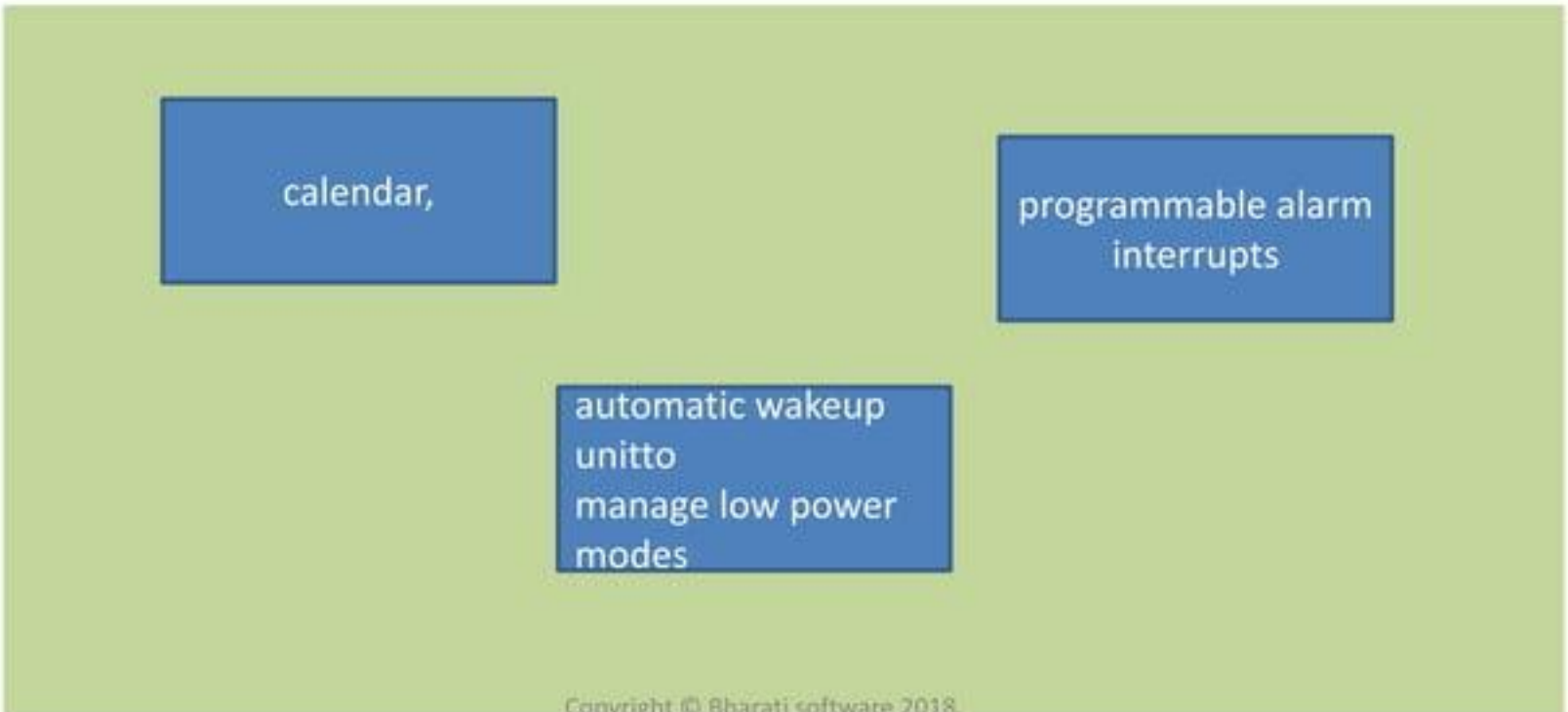
RTC Clock Sources

Figure 4. STM32F2xx or STM32F4xx RTC clock sources



The RTC features several prescalers that allow delivering a 1 Hz clock to calendar unit,
regardless of the clock source.

RTC Features



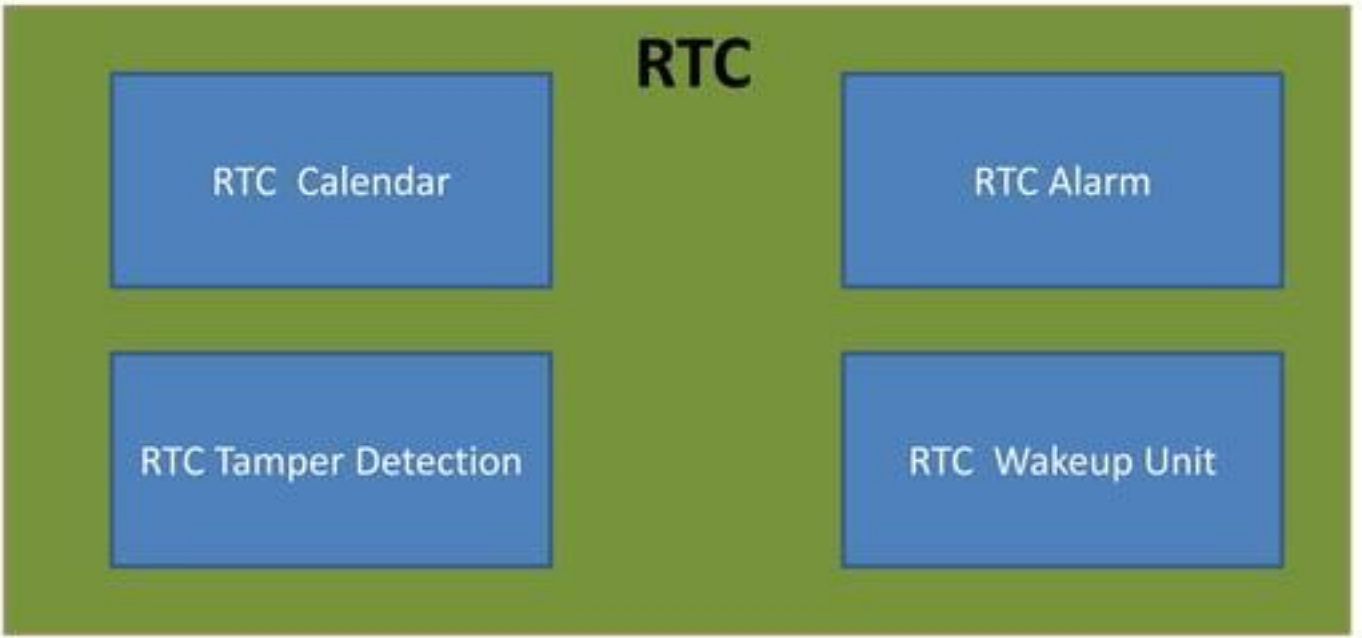
Backup SRAM

- Backup SRAM is part of the backup domain
- In stm32f446re MCU , there is 4KBs of Backup SRAM
- This is introduced in order to hold some data like the serial numbers, cryptographic keys, or any other application data, even if the Vdd is removed. (Supported by VBAT)
- It can be considered as an internal EEPROM when VBAT is always present.
- It is by default write protected (Actually all back domain peripherals are write protected , you need to first get the access in order to change any settings or content of the back up domain peripherals)

Backup SRAM

- Dedicated voltage regulator called **backup voltage regulator** is used to preserve the contents of the backup SRAM when in standby mode(if selected) or in VBAT modes

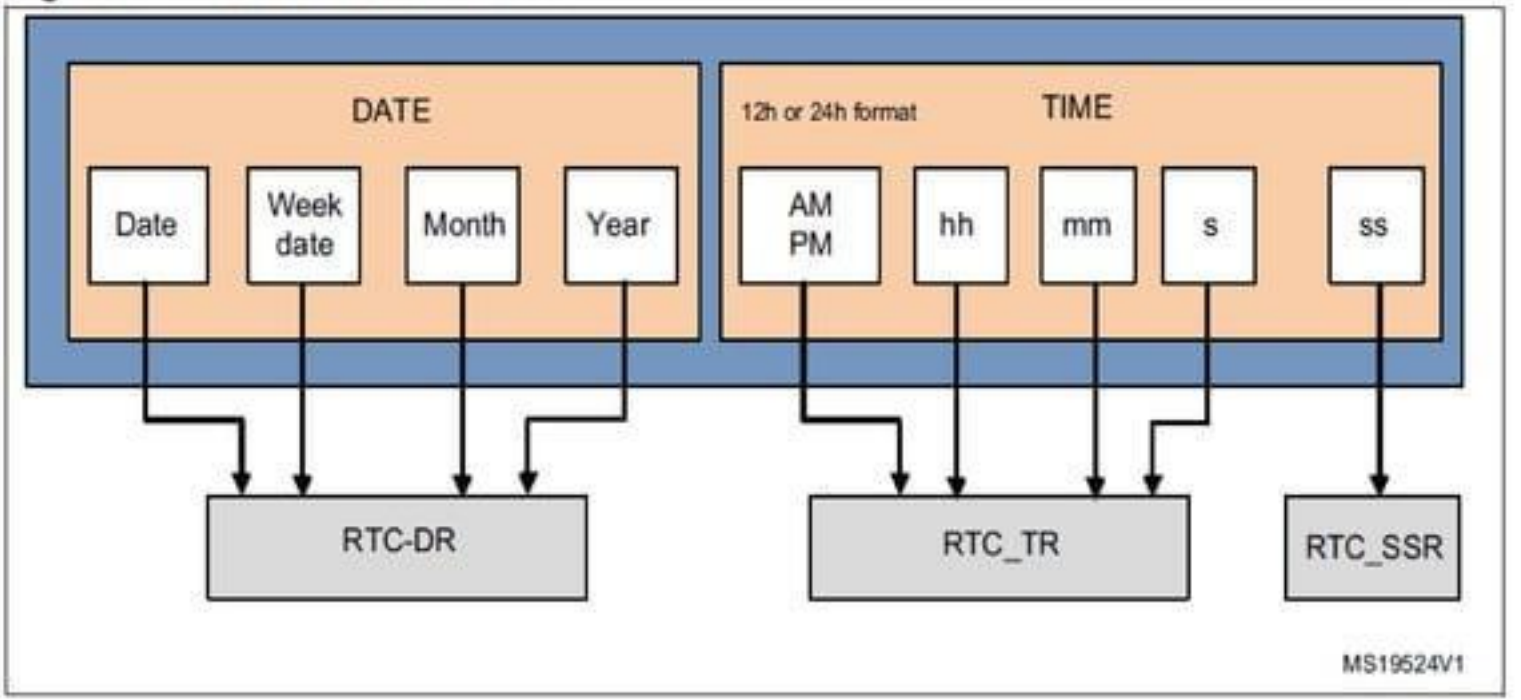
RTC Important features



RTC Calendar Unit

- A calendar keeps track of the time (hours, minutes and seconds) and date (day, week, month, year).
- Calendar comes with :
  - sub-seconds (not programmable)
  - seconds
  - minutes
  - hours in 12-hour or 24-hour format
  - day of the week (day)
  - day of the month (date)
  - Month
  - Year
- The STM32 RTC calendar is provided in BCD format
- Automatic management of 28-, 29- (leap year), 30-, and 31-day months

Figure 1. RTC calendar fields



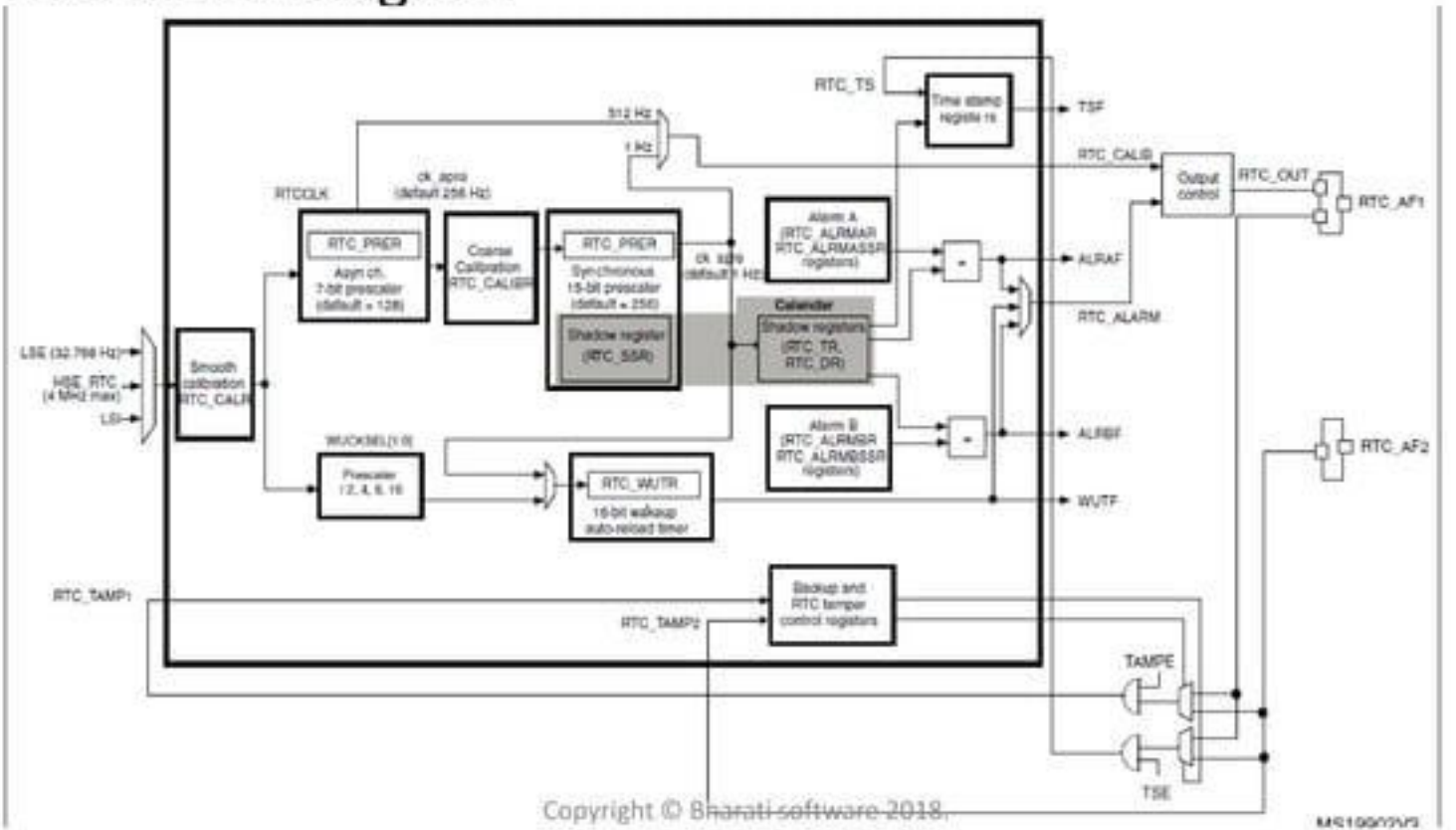
BCD vs Binary

Decimal	BCD	Binary
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
9	1001	1001
10	0001 0000	1010
15	00010101	1101
56	01011100	111000

RTC Calendar BCD fields Example

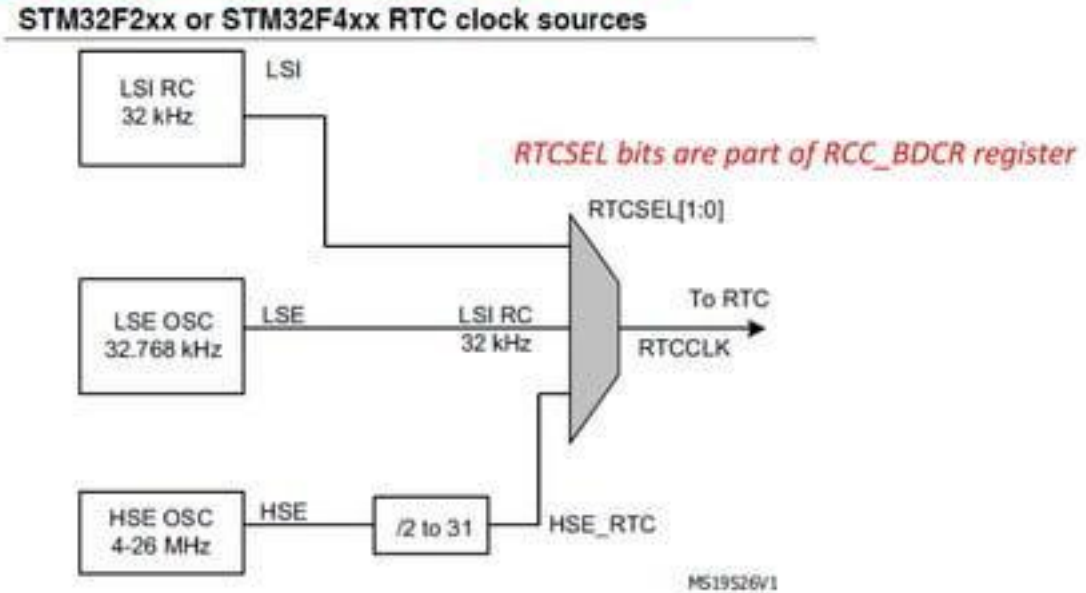
- Lets see an example
  - Program 2:40:58 AM in to RTC Time register
  - Program 12th June 2018 in to RTC Date register
- BCD of 2 ➔ 0010
- BCD of 40 ➔ 0100 0000
- BCD of 58 → 0101 1000
- BCD of 12 → 0001 0010
- BCD of 6(june) ➔ 0110
- Bcd of 2018 → 18 → 0001 1000

RTC Block Diagram



RTC Clock Source

The RTC can be driven by three clock sources LSE, LSI or HSE



Generating 1Hz from different clock sources

Remember calendar unit needs 1Hz clock to update time

Table 3. Calendar clock equal to 1 Hz with different clock sources

RTCCLK Clock source	Prescalers		ck_spre
	PREDIV_A[6:0]	PREDIV_S[12:0]	
HSE_RTC = 1MHz	124 (div125)	7999 (div8000)	1 Hz
LSE = 32.768 kHz	127 (div128)	255 (div256)	1 Hz
LSI = 32 kHz(1)	127 (div128)	249 (div250)	1 Hz
LSI = 37 kHz(2)	124 (div125)	295 (div296)	1 Hz

1. For STM32L1xx, LSI = 37 KHz, but LSI accuracy is not suitable for calendar application.
2. For STM32F2xx and STM32F4xx, LSI = 32 KHz, but LSI accuracy is not suitable for calendar application.

RTC and STM32 Cube

1. RTC Clock Selection
2. RTC Init
3. RTC Set Time and Date
4. RTC Get Time and Date
5. RTC Set/Get Alarm
6. RTC Deactivate Alarm
7. RTC Set/Deactivate Wakeup Timer

And various others.

RTC Clock Selection Procedure

1. Turn on the Required Clock (HSE/LSI/LSE)

For this we use :

HAL_RCC_OscConfig(RCC_OscInitTypeDef *RCC_OscInitStruct)

2. Select the RTCCLK source as HSE/LSI/LSE in RCC_BDCR register

For this we use :

HAL_RCCEx_PeriphCLKConfig(RCC_PeriphCLKInitTypeDef *PeriphClkInit)

RTC Init Procedure

For RTC Init, STM32 Cube layer provides below handle structure and API

HAL_RTC_Init(RTC_HandleTypeDef *hrtc)

Exercise

- Write a program to set the current time, day and Date information in to the RTC and then print back the Date, time, day information via UART whenever user button is pressed.

  Case 1 : Time format 12h

  Case 2 : Time format 24h

  Case 3 : Do a system reset and observe the effect on your configuration made to the RTC

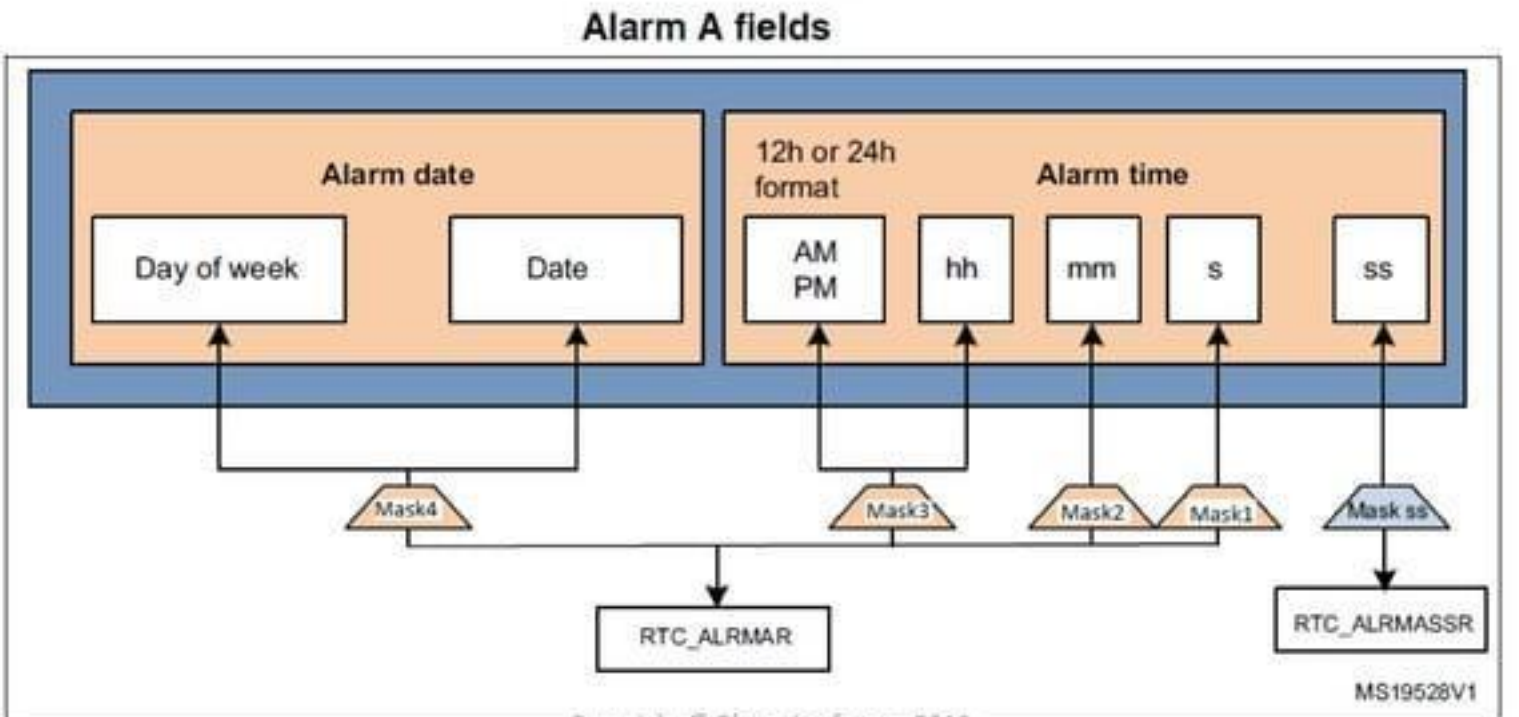
  Case 4 : See the effect on RTC when microcontroller exits Standby mode.

RTC Alarm

- STM32 RTC embeds two alarms, alarm A and alarm B, which are similar.

- An alarm can be generated at a given time or/and date programmed by the user.

RTC Alarm Fields



Example of RTC Alarm Masking

Example 1 :

Set Alarm @ 23:15:07 Every day

The above alarm means that,

1. We want alarm everyday , so date or day is don't care(mask it for comparison)
2. RTC calendar must be configured in 24h format
3. Alarm must be generated when calendar fields Hh:mm:ss exactly match with configured alarm fields

Example of RTC Alarm Masking

Example 2 :

Set Alarm @8AM on Every Sunday

The above alarm means that,

1. We want alarm only on Sunday
2. We are interested to compare day not date, so **WDSEL** bit of **RTC_ALRMAR** has to be 0
3. RTC calendar must be configured in 12h format
4. Alarm minutes and seconds fields are don't care (so just mask them for comparison)

Example of RTC Alarm Masking

Example 3 :

Set Alarm @ XX:45:09

The above alarm means that,

1. We want alarm at 45mins , 09 seconds every hour
2. Hour field is don't care
3. Date/Day field is don't care

RTC Alarm and STM32 Cube APIs

Exercise

- Write a program to Set an Alarm @ xx:45:09
  - An interrupt must be triggered during alarm
  - Use buzzer(you can turn on led if you don't have buzzer) to indicate the alarm
  - Configure the RTC calendar and Alarm in Button ISR

Exercise

- Write a program to Set an Alarm @ 12:00:15 PM Every day
  - An interrupt must be triggered during alarm
  - Use buzzer(you can turn on led if you don't have a buzzer) to indicate the alarm
  - Configure the RTC calendar and Alarm in Button ISR

Exercise

- Write a program to Set an Alarm @ 8AM Every Sunday
  - An interrupt must be triggered during alarm
  - Use buzzer(you can turn on led if you don't have a buzzer) to indicate the alarm
  - Configure the RTC calendar and Alarm in Button ISR

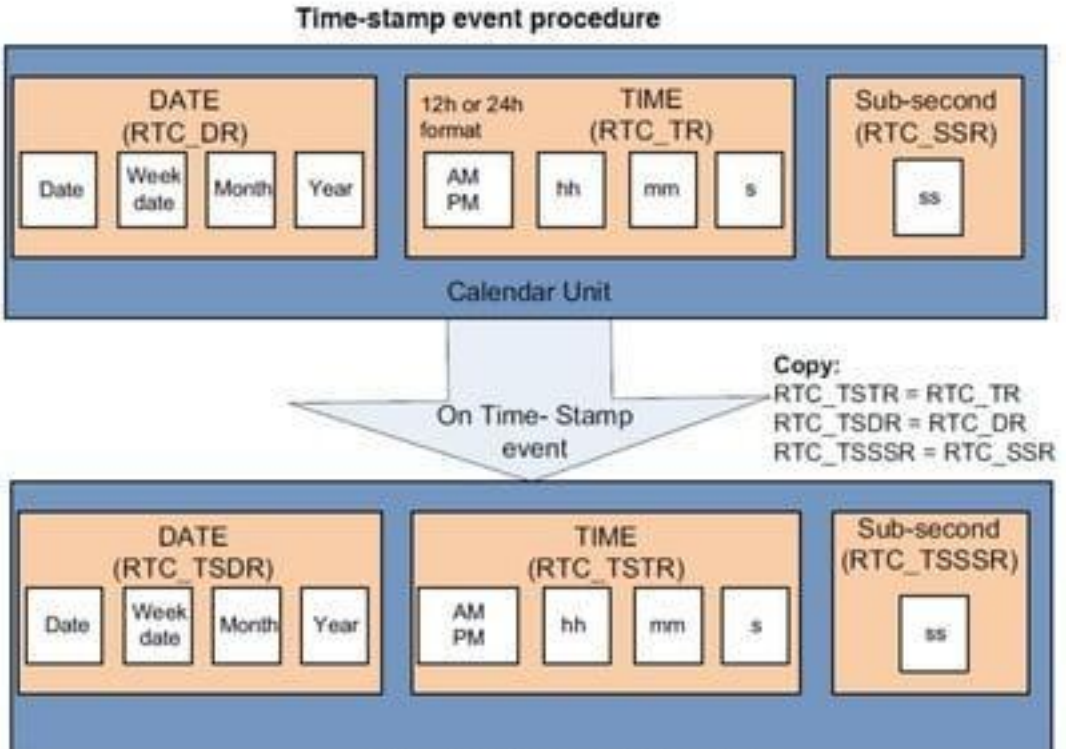
Exercise

- Write a program to wake up the MCU from STANDBY mode @ 23:15:30 Every day
  - Use RTC Alarm Interrupt as the wakeup source
  - Use buzzer(you can turn on led if you don't have a buzzer) to indicate STANDBY exit

Exercise

- Write a program to wake up the MCU from STOP mode @ 23:15:30 Every day
  - Use RTC Alarm Interrupt as the wakeup source
  - Use buzzer(you can turn on led if you don't have a buzzer) to indicate STOP mode exit
  - Configure the RTC calendar and Alarm in Button ISR

RTC Time Stamp Feature



Exercise

- Write a program which prints the time stamp of the LED ON Event by TIMER's time base interrupt
  - Use Timestamp feature of the RTC
  - Use RTC Time Stamp interrupt to print the time stamp details .
  - Use TIM6 time base generation for every 5 sec

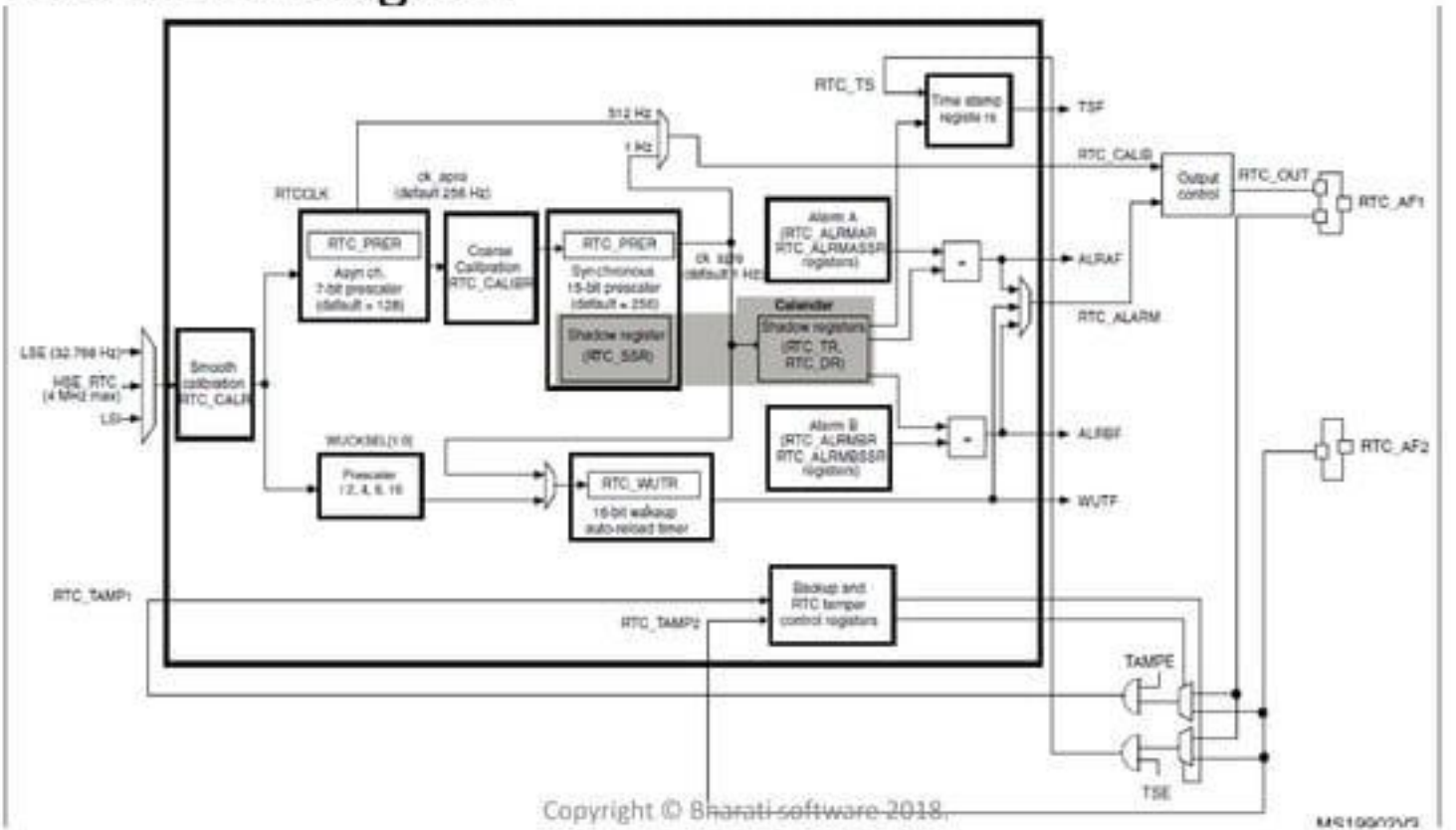
RTC periodic wakeup unit

- The STM32 features a periodic timebase and wakeup unit that can wake up the system when the STM32 operates in low power modes.
- This unit is a programmable down-counting auto-reload timer.
- When this counter reaches zero, a flag and an interrupt (if enabled) are generated.

Wakeup Unit Features

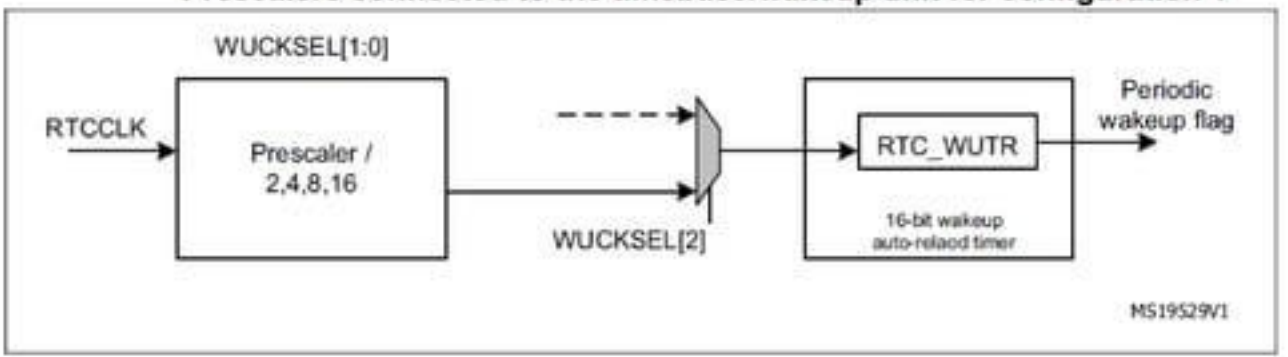
- It has Programmable down-counting auto-reload timer.
- Specific flag and interrupt capable of waking up the device from low power modes
- Wakeup alternate function output which can be routed to RTC_ALARM output (unique pad for alarm A, alarm B or Wakeup events) with configurable polarity.
- A full set of prescalers to select the desired waiting period.

RTC Block Diagram



Wakeup Timer Clock Config-1

Prescalers connected to the timebase/wakeup unit for configuration 1

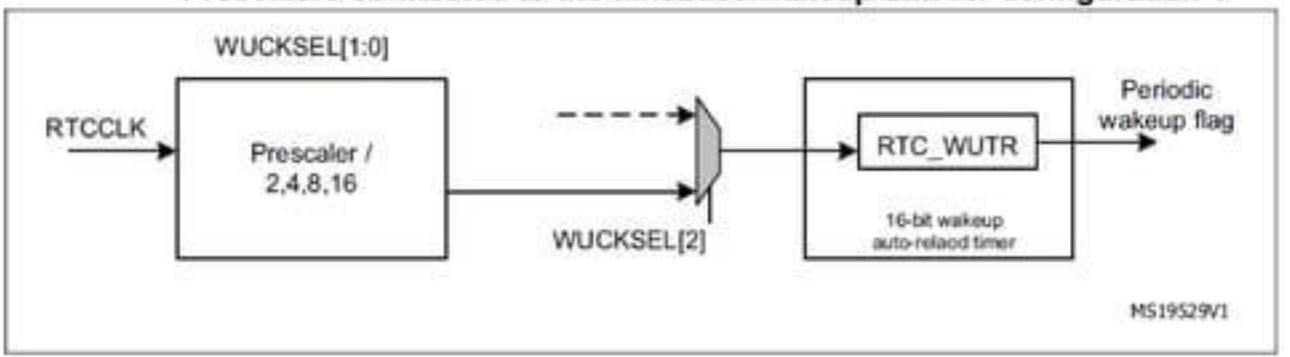


Timebase/wakeup unit period resolution with clock configuration 1

Clock source	Wakeup period resolution	
	WUCKSEL[2:0] = 000b (div16)	WUCKSEL[2:0] = 011b (div2)
LSE = 32 768 Hz	488.28 µs	61.035 µs

Wakeup Timer Clock Config

Prescalers connected to the timebase/wakeup unit for configuration 1



Timebase/wakeup unit period resolution with clock configuration 1

Clock source	Wakeup period resolution	
	WUCKSEL[2:0] = 000b (div16)	WUCKSEL[2:0] = 011b (div2)
LSE = 32 768 Hz	488.28 µs	61.035 µs