USB Keyboard Development using STM32 Micro-controller

By
ABHISHEK M PATIL
ACG Dept.
Feb 2016

CONTENTS

1. USB terms used

1. **Host**: PC with suitable OS
2. **Host controller**: It is the one which formats data for transmitting onto bus and translates received data to a format OS can understand.
3. **Root Hub**: It has one or more connectors for attaching devices.
   (Together Host controller and Root hub enable the OS to communicate with devices on the bus).
4. **Function**: It is a device that provides capabilities to the device.
   (Ex: keyboard, mouse)
5. **Hub**: It is a device which contains one or more connectors or internal connections to USB devices along with the hardware to enable communicating with the device.
6. **Device/Peripheral**: It is something you attach to the USB port on a PC or hub.

2. INTRODUCTION

The PS/2 interface is disappearing from the new generation PCs being replaced by the USB interface, which has become the standard interface between the PCs and peripherals. This change must be followed by keyboard designers, who must integrate the USB interface to connect the keyboard to the host system.

This provides the aim for this project to replace the PS/2 interface keyboard used in the ACE DIGI CNC System with the USB interface.

In the first part an overview of USB protocol is given and USB HID class device (keyboard) is explained in detail. Then USB Software model and the data transfer process for sending key press data from keyboard through USB interface is given.

In later parts of the document the process of developing the software required for the application is explained. A STM-32 Performance Stick tool is used as debugger system to develop and debug the firmware required for the application. Micro-controller architecture is explained and initial micro-controller configurations required for the application is discussed.

The steps required for tool to be configured as USB HID device and keypad interfacing to the micro-controller and overall program flow is also discussed.

For further development lower end micro-controller suggestions and components required for circuit schematic and procedure for loading the binary file to micro-controller is explained.

3. USB HID BASICS

The Universal Serial Bus (USB) is a fast and flexible interface for connecting devices to computers. Every new PC has at least a couple of USB ports. The interface is versatile enough to use with standard peripherals like keyboards and disk drives as well as more specialized devices, including one-of-a-kind designs. USB is designed from the ground up to be easy for end users, with no user configuring required in hardware or software.

USB is an interface that connects a device to a computer. With this connection, the computer sends or retrieves data from the device. USB gives developers a standard interface to use in many different types of applications. A USB device is easy to connect and use because of a systematic design process.

USB History

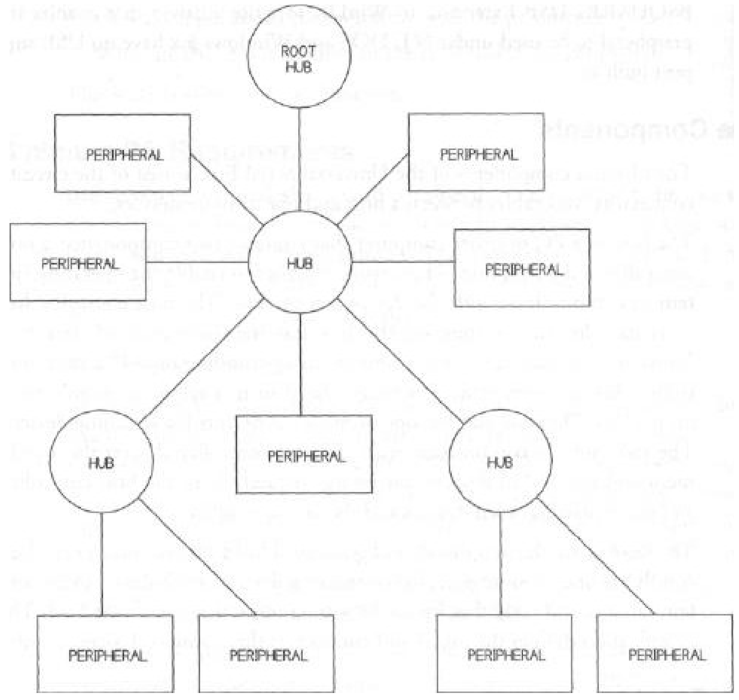
USB is an industry standard developed for the connection of electronic peripherals such as keyboard, mice, modems, and hard drives to a computer. This standard was developed in order to replace larger and slower connections such as serial and parallel ports. The standard was developed through a joint effort, starting in 1994, between Compaq, DEC, IBM, Intel, Microsoft, NEC, and Nortel. The goals were to develop a single interface that could be used across multiple devices, eliminate the many different connectors currently available at the time, and increase the data throughput of electronic devices.

USB Overview

USB systems consist of a host, which is typically a personal computer (PC) and multiple peripheral devices connected through a tiered-star topology. This topology may also include hubs that allow additional connection points to the USB system. The host itself contains two components, the host controller and the root hub. The host controller is a hardware chip-set with a software driver layer that is responsible for these tasks:

- Detect attachment and removal of USB devices
- Manage data flow between host and devices
- Provide and manage power to attached devices
- Monitor activity on the bus

At least one host controller is present in a host and it is possible to have more than one host controller. Each controller allows connection of up to 127 devices with the use of external USB hubs. The root hub is an internal hub that connects to the host controller(s) and acts as the first interface layer to the USB in a system.



USB protocols can configure devices at start-up or when they are plugged in at run time. These devices are broken into various device classes such as HID, Mass storage, Audio etc. Each device class defines the common behavior and protocols for devices that serve similar functions.

The HID class consists primarily of devices that are used by humans to control the operation of computer systems.

Typical examples of HID class devices include:

1. **Keyboards and pointing devices** – Standard mouse devices, trackballs, and joysticks.
2. **Front-panel controls** – Knobs, switches, buttons, and sliders.

Each device is given an address by the host, which is used in the data communication between that device and the host. USB device communication is done through pipes. These pipes are a connection pathway from the host controller to an addressable buffer called an endpoint. An endpoint stores received data from the host and holds the data that is waiting to transmit to the host. A USB device can have multiple endpoints and each endpoint has a pipe associated with it.

There are two types of pipes in a USB system, control pipes and data pipes. The USB specification defines four different data transfer types. Which pipe is used depends on the data transfer type.

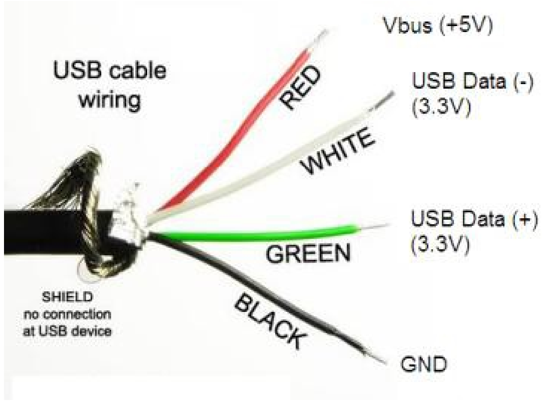
1. **Control Transfers –** Used for sending commands to the device, make inquiries, and configure the device. This transfer uses the control pipe.
2. **Interrupt Transfers –** Used for sending small amounts of bursty data that requires a guaranteed minimum latency. This transfer uses a data pipe.
3. **Bulk Transfers –** Used for large data transfers that use all available USB bandwidth with no guarantee on transfer speed or latency. This transfer uses a data pipe.
4. **Isochronous Transfers –** Used for data that requires a guaranteed data delivery rate. Isochronous transfers are capable of this guaranteed delivery time due to their guaranteed latency, guaranteed bus bandwidth, and lack of error correction. Without the error correction, there is no halt in transmission while packets containing errors are resent. This transfer uses a data pipe.

Every device has a control pipe and it is through this pipe that control transfers to send and receive messages from the device are performed.

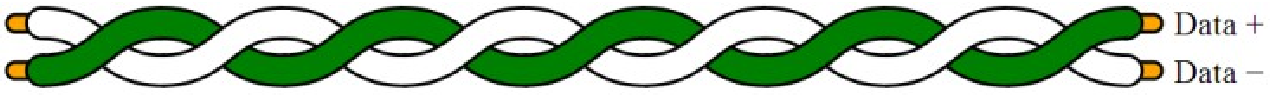
For HID class devices such as keyboard Interrupt Transfers are made using Endpoint1.

4. PHYSICAL INTERFACE

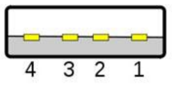
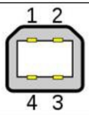
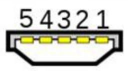
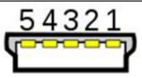
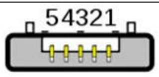
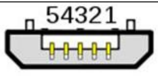
From a high-level overview, the physical interface of USB has two components: cables and connectors. These connectors connect devices to a host.
A USB cable consists of multiple components that are protected by an insulating jacket. Underneath the jacket is an outer shield that contains a copper braid. Inside the outer shield are multiple wires: a copper drain wire, a VBUS wire (red), and a ground wire (black). An inner shield made of aluminum contains a twisted pair of data wires.



In Full-Speed and High-Speed devices, the maximum cable length is 5 meters. To increase the distance between the host and a device, you must use a series of hubs and 5-meter cables. While USB extension cables exist in the market, using them to exceed 5 meters is against the USB specification. Low-Speed devices have slightly different specifications. Their cable length is limited to 3 meters and Low-Speed cables are not required to be a twisted pair



The upstream connection always uses a Type A port and connector, while the device uses Type B ports and connectors.

Type	Port Image	Connector Image
Type A		
Type B		
Mini-AB		
Mini-B		
Micro-AB		
Micro-B		

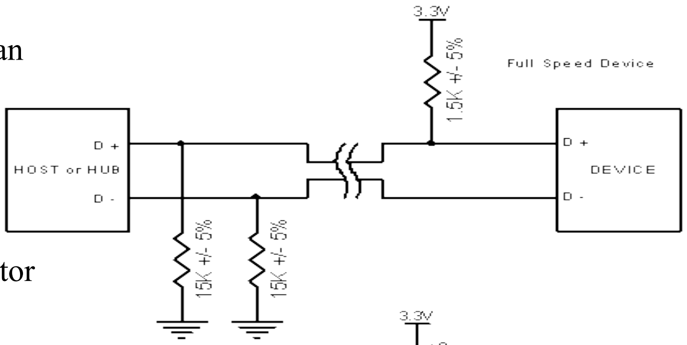
Speed

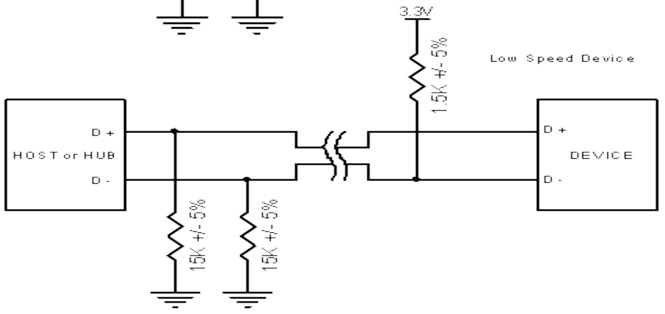
Three transfer speeds:

1. Low Speed – 1.5 Mbps (USB 1.1 and 2.0)
2. Full Speed – 12 Mbps (USB 1.1 and 2.0)
3. Hi-Speed – 480 Mbps (USB 2.0 only)

ATTACH EVENT

When we plug the device together with the host it is called an Attach Event (similarly, when we disconnect it is called a detach event). If you look at the initial condition for the bus with no devices attached you will notice that the D+ and D- are at the same 0 V potential because of the 15 K Ohm resistors found on the host side. When the cable is plugged in an endpoint device will provide a pull up resistor on either D+ or D depending on its speed capabilities. For a full speed device, the pull-up is attached to the D+ signal. When the host detects this voltage change it will begin what is called enumeration process at the full speed rate. If the voltage change is detected on the D- line, then the enumeration happens at low speed. This attach event is what signals the host that there is a new device attached to the bus.

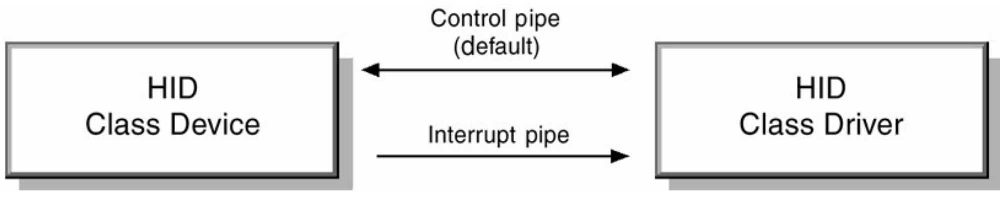




5. USB HID Descriptors

When a device is connected to a USB host, the device gives information to the host about its capabilities and power requirements. The device typically gives this information through a descriptor table that is part of its firmware. A descriptor table is a structured sequence of values that describe the device; these values are defined by the developer. All descriptor tables have a standard set of information that describes the device attributes and power requirements. If a design conforms to the requirement of a particular USB device class, additional descriptor information that the class must have is included in the device descriptor structure.

A USB/HID class device uses a corresponding HID class driver to retrieve and route all data. The routing and retrieval of data is accomplished by examining the descriptors of the device and the data it provides.



Device Descriptor

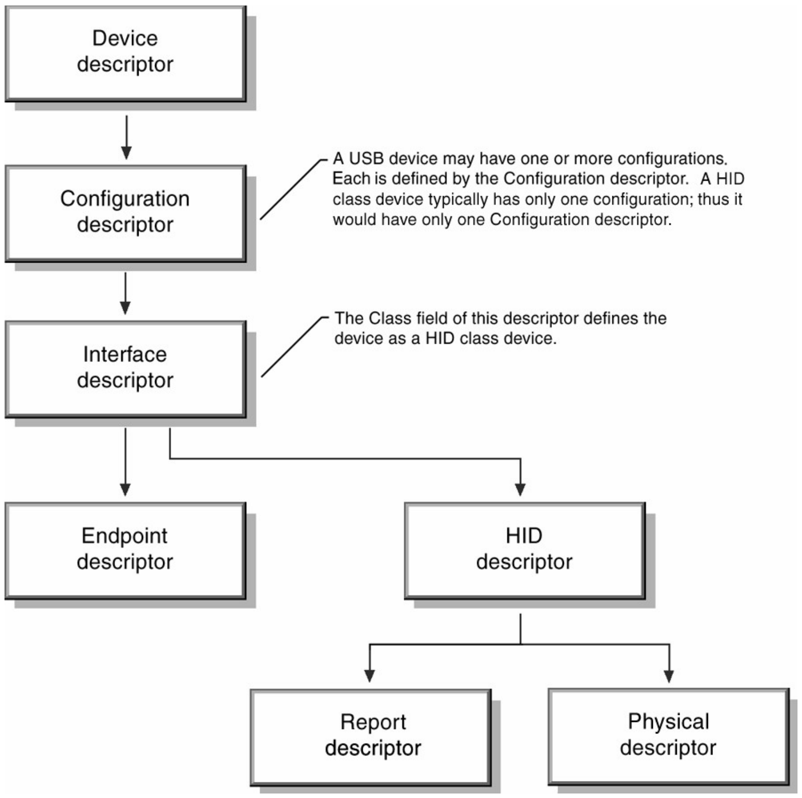
Device descriptors give the host information such as the USB specification to which the device conforms, the number of device configurations, and protocols supported by the device, Vendor Identification (also known as VID, which is something that each company gets uniquely from the USB Implementers Forum), Product Identification, and a serial number if the device has one. The device descriptor is where some of the most crucial information about the USB device is contained.

Configuration Descriptor
This descriptor gives information about a specific device configuration such as the number of interfaces, if the device is bus-powered or self-powered, if the device can start a remote wake-up, and how much power the device needs.

Interface Descriptor
An interface descriptor describes a specific interface within a configuration. The number of endpoints for an interface is identified in this descriptor. The interface descriptor is also where the USB Class of the device is declared. There are many predefined classes that a USB device can be, A USB device class identifies the device functionality and aids in the loading of a proper driver for that specific functionality.

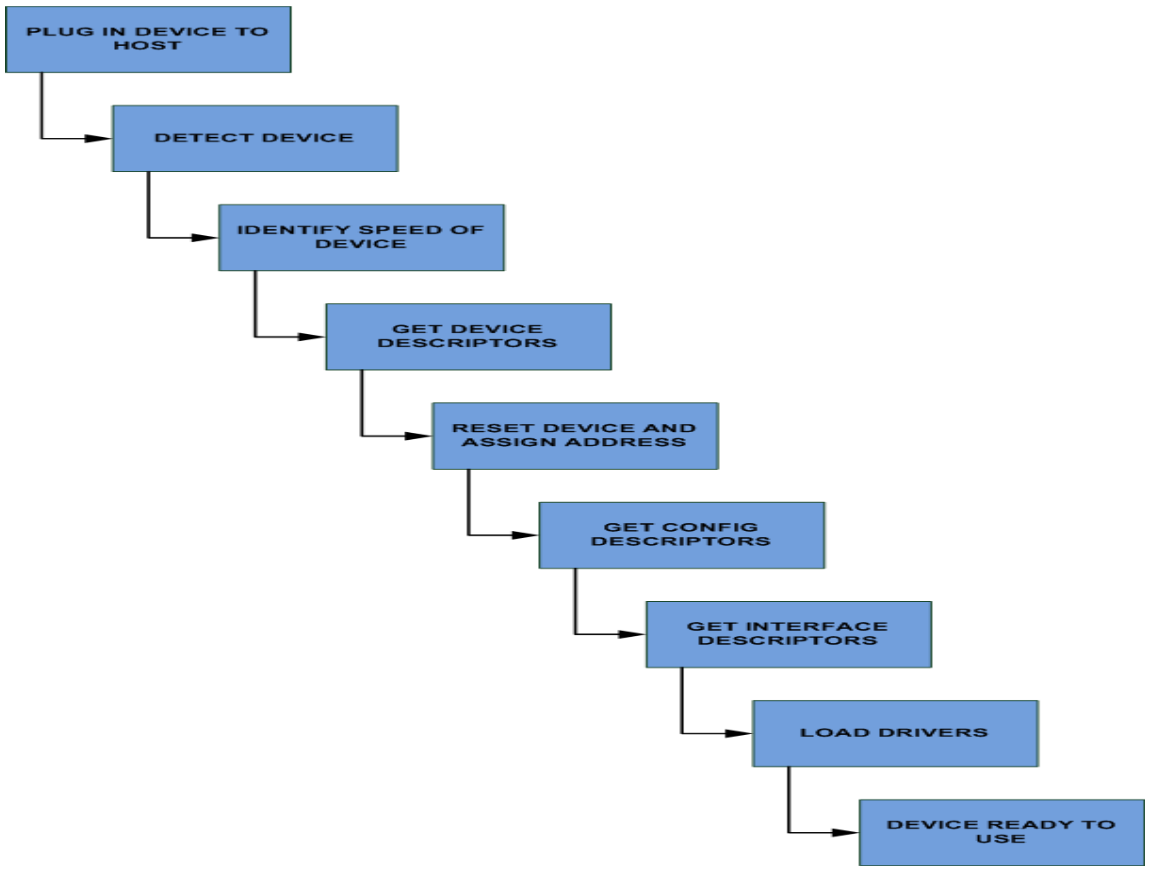
Endpoint Descriptor
Each endpoint used in a device has its own descriptor. This descriptor gives the endpoint information that the host must have. This information includes direction of the endpoint, transfer type, and maximum packet size.

String Descriptor
The string descriptor is another optional descriptor and gives user readable information about the device. Possible information contained in the descriptor is the name of the device, the manufacturer, the serial number, or names for the various interfaces or configurations. If strings are not used in a device, any string index field of the descriptors mentioned earlier must be set to 00h.


Device descriptor
Configuration descriptor
A USB device may have one or more configurations. Each is defined by the Configuration descriptor. A HID class device typically has only one configuration; thus it would have only one Configuration descriptor.
Interface descriptor
The Class field of this descriptor defines the device as a HID class device.
Endpoint descriptor
HID descriptor
Report descriptor
Physical descriptor

The descriptor values used for this project are explained in the section **STM32 USB PREPARATION**.

6. ENUMERATION PROCESS



Following the initial attachment of a USB I/O device on to a Host PC, a long conversation is held on this control endpoint so that the operating system can integrate the I/O device into its operating environment. This process is called **Enumeration** in the USB literature and involves a series of pre-formatted standard USB requests.

After the basic initialization, the USB module is now able to react with an interrupt on packets, addressed to the control endpoint EP0. Enumeration is the process in which the configuration is integrated into the USB system. The host is now able to assign a clear USB address to the device, located in the range from 1 to 127.

In another step in the enumeration, the host will request the configuration of the device. For this purpose descriptors are made available by the device, which contain information about the status and about one or more possible configurations. The host loads these descriptors, selects thereafter a suitable driver and forces the device to take a certain configuration. In consequence, the device will be ready for use and will be able to transfer data via the interrupt endpoint EP1.

Report Descriptor

The **Report** descriptor is unlike other descriptors in that it is not simply a table of values. The length and content of a Report descriptor vary depending on the number of data fields required for the device's report or reports. A report descriptor defines the format and uses of the data that carries out the purpose of the device. If the device is a keyboard, the data reports keystrokes.

The Report descriptor provides a description of the data provided by each control in a device. Each **Main** item tag (**Input**, **Output**, or **Feature**) identifies the size of the data returned by a particular control, and identifies whether the data is absolute or relative, and other pertinent information. A Report descriptor is the complete set of all items for a device. By looking at a Report descriptor alone, an application knows how to handle incoming data, as well as what the data could be used for.

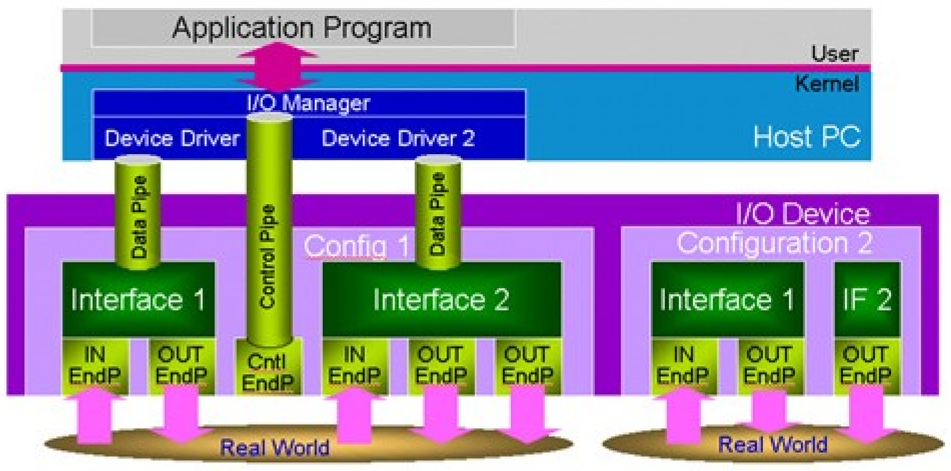
HID REPORT EXAMPLE (KEYBOARD)

```
0x05, 0x01,                    // Usage Page (Generic Desktop)
0x09, 0x06,                    // Usage (Keyboard)
0xA1, 0x01,                    // Collection (Application)
0x75, 0x01,                    // Report Size (1)
0x95, 0x08,                    // Report Count (8)
0x05, 0x07,                    // Usage Page (Key Codes)
0x19, 0xE0,                    // Usage Minimum (224)
0x29, 0xE7,                    // Usage Maximum (231)
0x15, 0x00,                    // Logical Minimum (0)
0x25, 0x01,                    // Logical Maximum (1)
0x81, 0x02,                    // Input (Data, Variable, Absolute) -- Modifier byte
0x95, 0x01,                    // Report Count (1)
0x75, 0x08,                    // Report Size (8)
0x81, 0x03,                        // (81 01) Input (Constant) -- Reserved byte
0x95, 0x06,                        // Report Count (6)
0x75, 0x08,                        // Report Size (8)
0x15, 0x00,                        // Logical Minimum (0)
0x25, 0x66,                        // Logical Maximum(102)
0x05, 0x07,                        // Usage Page (Key Codes)
0x19, 0x00,                        // Usage Minimum (0)
0x29, 0x66,                        // Usage Maximum (102)
0x81, 0x00,                        // Input (Data, Array) -- Key arrays (6 bytes)
0xC0                          // End Collection
```

This values for the Report Descriptor is used in this project.

Note: This Report descriptor is only for IN reports. To add OUT reports like Caps lock LED further fields are required.

7. USB Software Model



A USB I/O device is a combination of software and hardware. An I/O device interfaces to the real world, data collected from the real world is placed in a local buffer area called an **IN endpoint.** the Host PC will collect this data later. Similarly, data is delivered by the host PC into a local buffer called an **OUT endpoint,** this data is then distributed to the real world by the I/O device.

A collection of endpoints is called an **Interface**. An interface describes the device **Class**. A typical operating system will include a range of class drivers, such as printer, audio, human interface device (HID) and mass storage, to support a wide collection of I/O devices. There is a one-to-one mapping of I/O device interface to Host PC device driver. The INF file mechanism is used in the Windows operating system to bind an interface to a device driver.

An I/O device also includes a C**ontrol endpoint**. This endpoint is bi-directional and is used by the operating system to discover the identity and capabilities of the I/O device and also to control its operation.

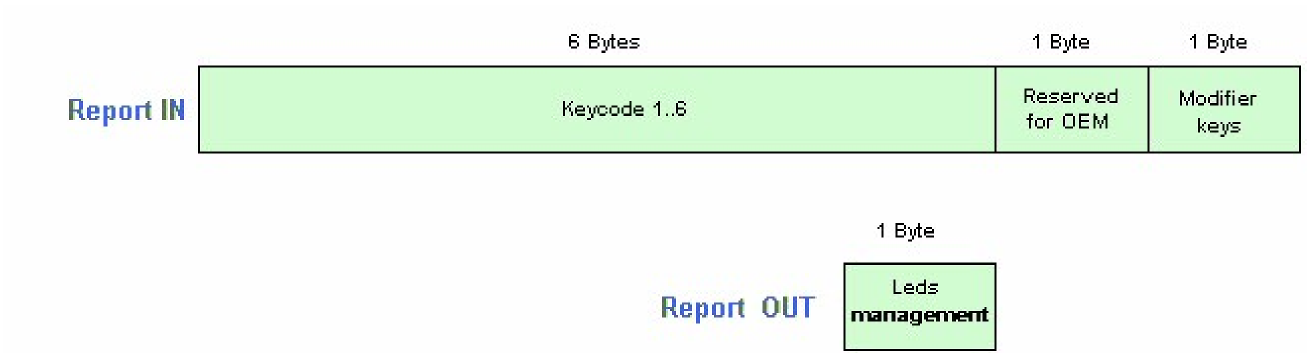
Interrupt transfer is used to send data from device to host in HID class devices. Although the name interrupt transfer suggests that a device can cause hardware interrupt that results in faster response from the Host, but interrupt transfer occur only when host polls a device. The transfers are interrupt like, because they guarantee that the host will request data with minimum delay.

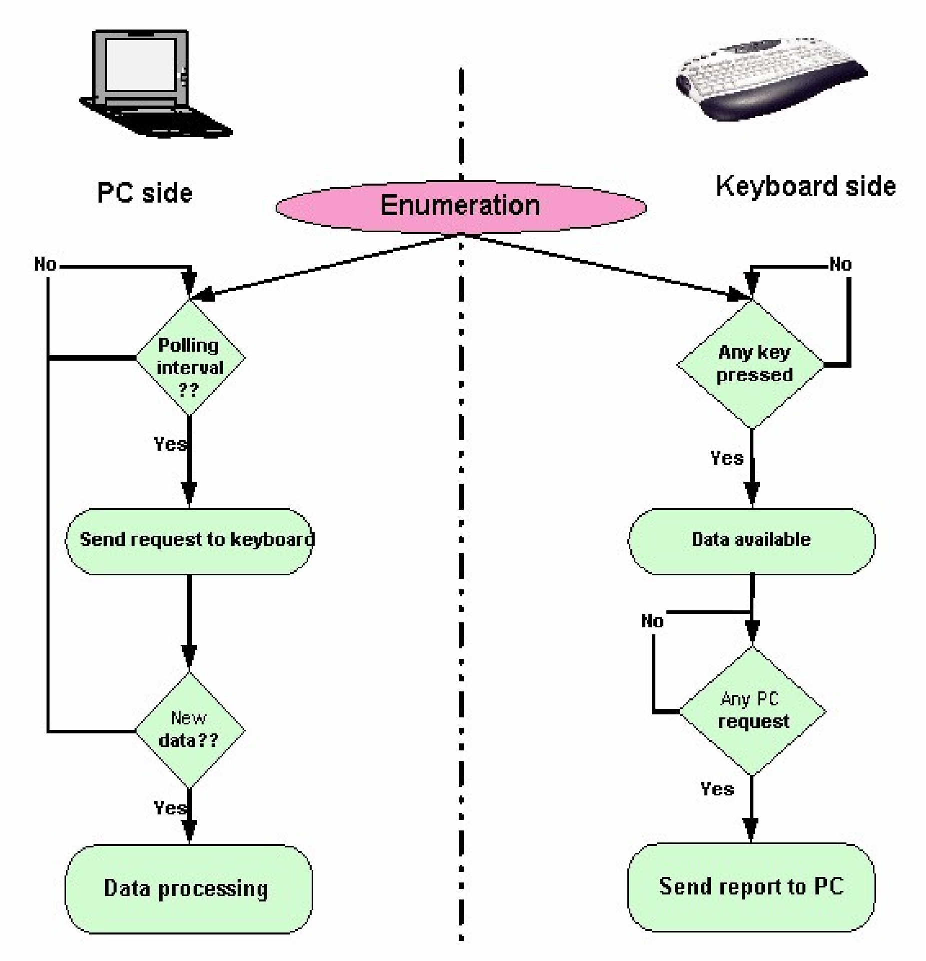
8. Data Transfer Process

The firmware periodically writes scan patterns to the scan matrix columns, and reads the row result to determine which keys are pressed. The scan codes of the keys pressed are sent to the host using IN endpoint.

The PC asks the keyboard if there is new data available each P time (polling interval time), That polling interval can be specified in the endpoint descriptor. The keyboard will send the data if it is available, otherwise, it will send a NAK (No Acknowledge) to tell the PC that there is no data available.

The data exchanges between the PC and the keyboard are called reports. The report which contains the keys pressed is the report IN (Keyboard to PC). The report which contains the LED's status (NUM LOCK, CAPS LOCK, SCROLL LOCK...) is the report OUT (PC to Keyboard) (Alternatively control endpoint can be used to receive OUT report).





9. INPUT REPORT EXAMPLE (KEYBOARD)

A transfer that carries data to the host from the device is known as an input transfer. The eight bytes that are sent to the host are organized as modifier keys followed by general keys. A modifier key is a key that modifies the function of a general key. While the information is sent in an 8-byte packet, there are only seven bytes of useful information. This is because of the reserved byte (byte 1).The reserved key may be used on a keyboard that contains a nonstandard key that performs a function that is specific to that PC.

The last six bytes in the configuration are the general keys. As many as six key codes can be sent to the PC in a given transaction. This enables as many as six simultaneous key presses. The order of the key codes in the array is not significant.

Input report for a character 'A'

Byte	**Value**	**Info**
0	0010 0000	(Right Shift)
1	Reserved	(For OEM Use)
2	0000 0100	(“A”)
3	0000 0000	NA
4	0000 0000	NA
5	0000 0000	NA
6	0000 0000	NA
7	0000 0000	NA

Modifier keys values

Bit	**Mod Value**	**Key**
0	0000 0001	Left Ctrl
1	0000 0010	Left Shift
2	0000 0100	Left Alt
3	0000 1000	Left GUI (Win/Apple)
4	0001 0000	Right Ctrl
5	0010 0000	Right Shift
6	0100 0000	Right Alt
7	1000 0000	Right GUI (Win/Apple)

Each modifier key has a corresponding bit associated with it. Although usage codes are defined in the usage table as (E0-E7), the usage is not sent as array data. This means that the modifier keys are stored in a bit field of information. The modifier keys are sent as variable data, which means that each individual bit in the 8-bit value corresponds to one of the modifier keys.

HID devices often have indicators that give the user status information about the host. With a keyboard, there may be LED indicators for the Num Lock, Caps Lock, or Function Lock keys. When a user presses any of the corresponding buttons such as Caps Lock or Num Lock, the host manages the updating of the LED information. Additionally, LED states are set by sending a report to the keyboard device via a SET_REPORT(Output). Rather than output data being transferred through a dedicated OUT endpoint, SET_REPORTs are done through Endpoint 0, the control endpoint, and do not require a dedicated endpoint to be provided.

10. GETTING USB VENDOR ID

To sell or market a USB product you must have a Vendor ID. This means that you must acquire one from USB-IF. It is important to make sure that you get a Vendor ID for your company to avoid legal consequences. There are three options on how to acquire a VID:

1. Vendor ID can be acquired by becoming USB-IF member. Membership fee is 4000$ annually.

2. By getting logo license and becoming eligible to use USB logo with products. Logo license fee 3500$ for two year term. In addition Vendor ID should be purchased at cost of 5000$.

3. Vendor ID without logo license agreement at the cost of 5000$. Not authorized to use the USB logo with the products.

Another company's Vendor ID can be used in development environment, but cannot be used in production design.

The main reason behind this is due to operating systems, such as Windows, remember the system files (inf file, driver, and so on) used by the device and loads them every time the device is attached. If any other company's VID is used by their customers, then the possibility that another customer may use the same VID/PID for their product exists. In this case, the end product may bind to a different driver and malfunction. Ultimately this causes trouble for the end customer.

11. STM-32 EVALUATION TOOL

For developing the USB keyboard application a STM32 Performance stick was used for evaluation purpose. The STM32-Performance Stick is a specific debugger system being able to emulate the integrated STM32F103RBT6 microcontroller with on-chip debug support. It provides a USB communication port for connecting the STM32-Performance Stick to a PC.

Requirements for using STM32 Performance Stick as USB keyboard:

Tools
- Hitex HiTOP tool chain for STM32-PerformanceStick including
- HiTOP5 IDE and debugger
- TASKING C-Compiler for ARM Cortex V2.0r2
- USB device driver
- STM32-PerformanceStick device with STM32F103RBT6 controller

Features and Components
- Platform: STM32F103RBT6 ARM Cortex-M3 core
- STM32-PerformanceStick device hardware
- Tasking C-Compiler for ARM Cortex version 2.0r2
- Featured STMicroelectronics library for STM32 is used

STM-32 Performance Stick (based on STM32F103RBT6 micro-controller) along with STM-32 IO board should be used for developing the application.

STM32F103RBT6 micro-controller specs:

- ARM 32bit Cortex-M3 CPU core
- 128 KB flash memory
- 20 KB SRAM
- 51 I/O pins
- USB 2.0 interface
- 72 MHz CPU speed

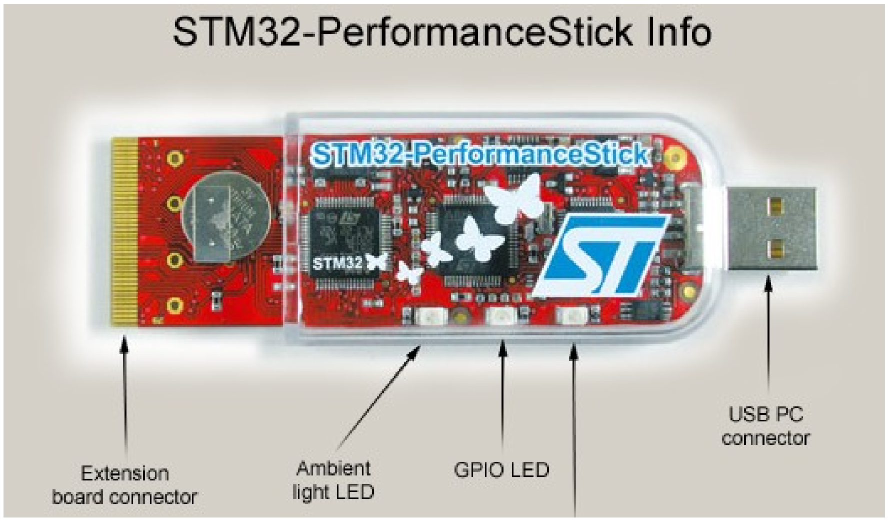
Software tools used along with it are:

- HiTop 5 IDE
- Dashboard GUI
- Task/GCC compilers

USB Joystick mouse demo example is provided along with it.

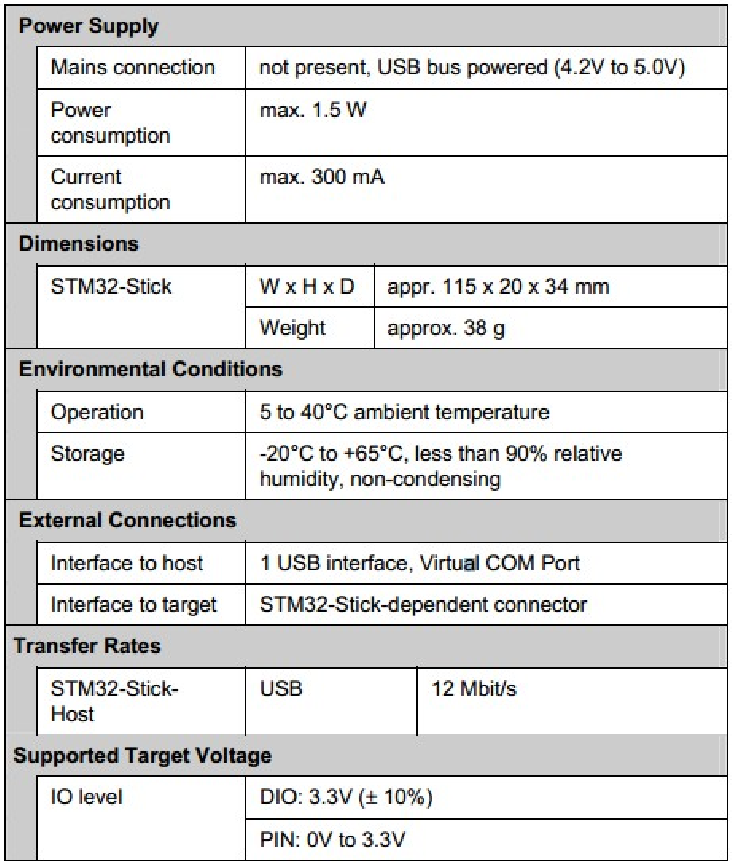
STM32-PerformanceStick:

The STM32-Performance Stick's USB interface is implemented in accordance with the USB specification 2.0.



As far as the transmission rate is concerned, the STM32-Performance Stick is a full-speed device. Since the STM32-Performance Stick is a USB-bus powered device, no external power supply is required. STM32-Performance Stick consumes up to 300 mA (depending on the extension board used) and thus requires a powered hub connection.

Technical specifications:



Power Supply		
Mains connection	not present, USB bus powered (4.2V to 5.0V)	
Power consumption	max. 1.5 W	
Current consumption	max. 300 mA	
Dimensions		
STM32-Stick	W x H x D	appr. 115 x 20 x 34 mm
	Weight	approx. 38 g
Environmental Conditions		
Operation	5 to 40°C ambient temperature	
Storage	-20°C to +65°C, less than 90% relative humidity, non-condensing	
External Connections		
Interface to host	1 USB interface, Virtual COM Port	
Interface to target	STM32-Stick-dependent connector	
Transfer Rates		
STM32-Stick-Host	USB	12 Mbit/s
Supported Target Voltage		
IO level	DIO: 3.3V (± 10%)	
	PIN: 0V to 3.3V	

STM32-IO board:

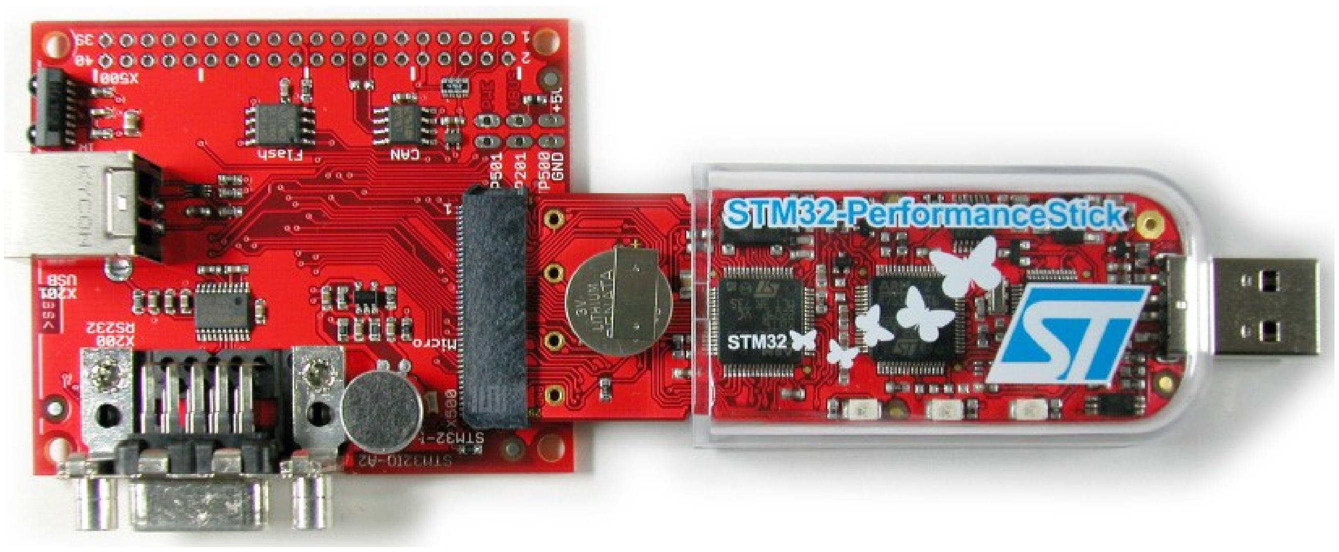
The STM32-IO-Board can be used together with the STM32-Performance Stick debugger system. It contains architecture- and device-specific information and all technical data of the system.

The STM32-Performance Stick is a specific debugger system being able to emulate the integrated STM32F103RBT6 micro controller with on-chip debug support.

It provides a USB communication port for connecting the STM32- Performance Stick to a PC.

The features of the STM32-IO-Board are summarized as follows:

- 2x40-pin 0.635mm pitch PCB extension Stick-I/O-connector for the
- STM32-PerformanceStick MEC6-140-02-L-D-RA1 (Samtec)
- 2x20-pin IO connector pads with reset and IRQ capability via jumpers
- USB-B connector with USB protection circuit USBLC6-2 (STM)
- RS232 connector with driver circuit ST3222 (STM)
- IrDA device TFDU4101 (Vishay)
- SPI Flash device M25PE16 (STM)
- CAN Transceiver circuit L9616 (STM)
- Microphone EMY-62MP or similar (EKULIT) with TSH300 OP amplifier (STM)
- USB-powered via stick or from external USB-B or IO- connector with jumper selection

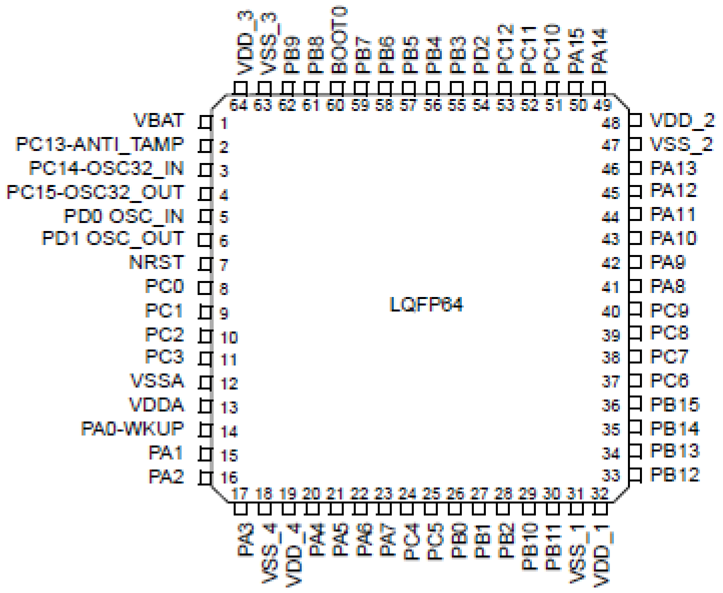


The STM32 Performance Stick contains STM32F103RBT6 micro-controller with other debugger circuits and is connected to the PC through USB COM port to load the program to the micontroller.

The STM32 IO board helps access to GPIO pins of micro-controller and has a USB connector to connect the system to the host.

12. MICROCONTROLLER (MCU)

The STM32F103xx Micro-controller is produced by *ST Company.* The family of STM32F103xx Microcontrollers consists of ARM Cortex-M3 32-bit RISC core, high speed embedded memories (Flash memory is up to 128 Kbytes and Static Random Access Memory (SRAM) is up to 20 Kbytes), I/Os (Input/Output) and peripherals which they are cooperating together by connecting to two APB (Advanced Peripheral Bus) buses. The STM32F103xx microcontroller includes many peripherals as well as two 12-bits ADCs, an Advanced Control Timer, three General Purpose 16-bit timers and also a PWM (Pulse Width Modulation) timer. It is also provided by two I²Cs (Inter-Integrated Circuit) and SPIs (Serial Peripheral Interface), three USARTs (Universal Synchronous / Asynchronous Receive Transmitter), an USB and a CAN (Controller Area Network) as the communication interface system.



STM32F103 Performance line Pinout

The medium-density microcontroller is used and it has 64 pins. This microcontroller family consists of three ports which PA, PB and PC are MCU ports and each port has 16 pins as I/Os. V_{SS}, V_{DD} and V_{BAT} are used to bias microcontroller by using external power supply.

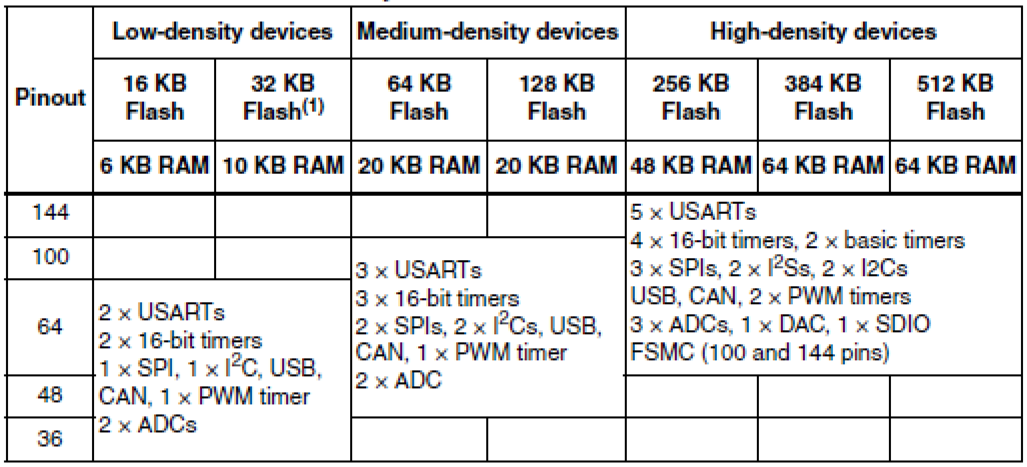
Family group of STM32F103

The STM32F103xx family micro-controllers are divided into three groups:

Low-density: The STM32F103x4 and STM32F103x6 are Low-density devices.

Medium-density: The STM32F103x8 and STM32F103xB are Medium-density devices.

High-density: The STM32F103xC, STM32F103xD and STM32F103xE are High- density devices.



Pinout	Low-density devices		Medium-density devices		High-density devices		
	16 KB Flash	32 KB Flash(1)	64 KB Flash	128 KB Flash	256 KB Flash	384 KB Flash	512 KB Flash
	6 KB RAM	10 KB RAM	20 KB RAM	20 KB RAM	48 KB RAM	64 KB RAM	64 KB RAM
144					5 × USARTs 4 × 16-bit timers, 2 × basic timers 3 × SPIs, 2 × I^{2}Ss, 2 × I2Cs USB, CAN, 2 × PWM timers 3 × ADCs, 1 × DAC, 1 × SDIO FSMC (100 and 144 pins)		
100			3 × USARTs 3 × 16-bit timers 2 × SPIs, 2 × I^{2}Cs, USB, CAN, 1 × PWM timer 2 × ADC				
64	2 × USARTs 2 × 16-bit timers 1 × SPI, 1 × I^{2}C, USB, CAN, 1 × PWM timer 2 × ADCs						
48							
36							

These three groups are made according to the feature of the microcontroller STM32F103xx family members:

Low-density microcontrollers have lower Flash memory and RAM (Random Access Memory), less timer and peripherals in compare to the other two groups. Thus, Medium-density and High-density consist of higher Flash memory, RAM capacities and also have more additional peripherals.

Low-density families include 16 KB to 32 KB Flash memory and 6 KB to 10 KB RAM capacities. They consist of 1 × CAN, 1 × USB, 1 × PWM timer, 1 × I²C, 1 × SPI and 2 × ADCs, 2 × USARTs, and 2 × 16-bits timers. The differences between Low-density families are regarding the number of their pinout packages. There are three kinds of pinout packages which they made up 36, 48 and 64 pins. Their Flash memory is improved from 64 KB to 128 KB and RAM capacity is 20 KB.

Medium-density families have more properties in compare to Low-density families. The number of peripherals and pinouts of them are improved. They have 1 × CAN, 1 × USB, 1 × PWM timer, 2 × I²C, 2 × SPI, 2 × ADCs, 3 × USARTs, and 3 × 16-bits timers. They also have three kinds of pinout packages that consist of 48, 64 and 100 pins.

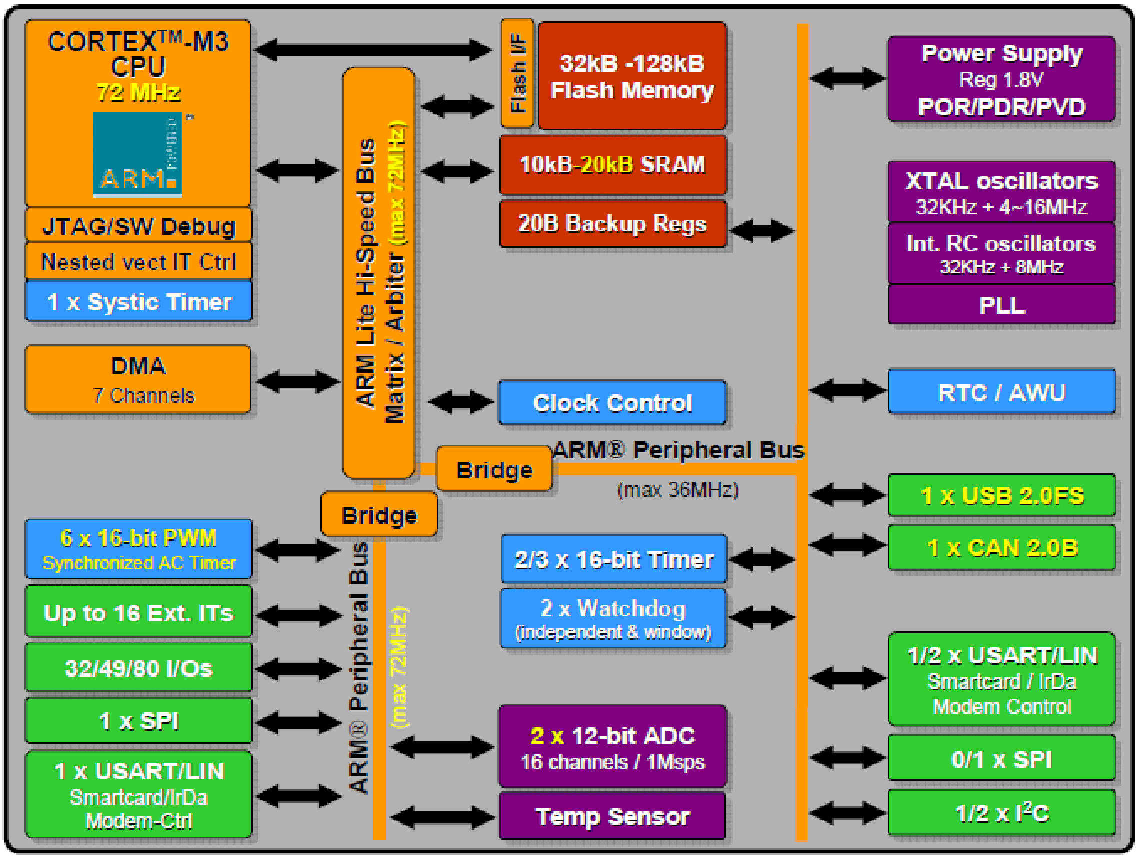
High-density families are completed more than others and have more peripherals. They made up these peripherals, such as 1 × CAN, 1 × USB, 1 × PWM timer, 2 × I²Ss (SPI), 2 × I²C, 3 × SPI, 2 × ADCs, 1 × DAC, 5 × USARTs, 2 × basic timers, 4 × 16-bits timers, 1 × SDIO (Secure Digital Input Output), and 1 ×FSMC (Flexible Static Memory Controller).

For this project the STM32F103xB microcontroller (Medium-density device) is applied and it should be mentioned that all STM32 families' performance line is fully compatible.

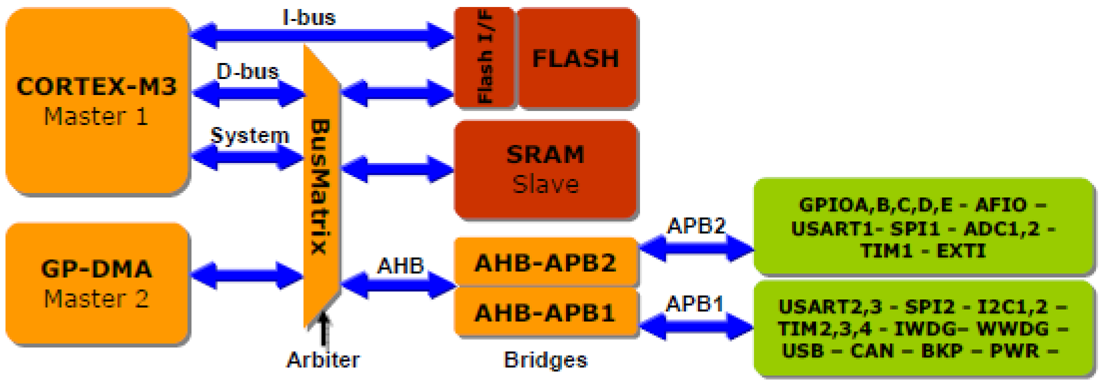
These three categories for the STM32F103xx microcontroller is useful for designers which they can select suitable family for their projects. This large variety of the STM32 causes to reduce the price of products and also products can have a better shape and size.

Components of STM32F103 MCU

This section explains the micro-controller core, memories, I/Os and peripherals which are introduced at the block diagram. There are a brief explanations for the STM32F103 micro-controller parts in the below.



System Architecture consists of Buses, and General purpose DMA (Direct Memory Access), Internal SRAM, Internal Flash Memory which some of them consider as masters and others consider as slaves. Figure 4 is shown bus Architecture for the STM32 family micro-controllers. The access between Core system bus and DMA bus are controlled by BusMatrix.


CORTEX-M3
Master 1
GP-DMA
Master 2
I-bus
D-bus
System
BusMatrix
Arbiter
AHB
Flash I/F
FLASH
SRAM
Slave
AHB-APB2
AHB-APB1
Bridges
APB2
APB1
GPIOA,B,C,D,E - AFIO - USART1- SPI1 - ADC1,2 - TIM1 - EXTI
USART2,3 - SPI2 - I2C1,2 - TIM2,3,4 - IWDG- WWDG - USB - CAN - BKP - PWR -

Masters:

I-bus (Cortex-M3 ICode bus): It connects the Cortex M3 core to the Flash memory instruction in order to do pre-fetching.
D-bus (DCode bus): It connects the Cortex-M3 core to the Flash memory Data interface.
S-bus (System bus): It connects the Cortex-M3 core peripheral bus to a BusMatrix in order to control the arbitration between the DMA and Core.
GP-DMA bus (General Purpose DMA): It connects CPU(Central Processing Unit) and DMA to the Flash memory, SRAM and Peripherals through BusMatrix in order to make communication between them.

Slaves:

Internal SRAM
Internal Flash Memory
AHB (Advanced High Performance Bus) to APB (Advanced Peripheral Bus) bridge: This bridge divides AHP bus into two buses, APB1 and APB2. APB1 is for peripheral which their frequency is 36 MHz and APB2 is for peripherals which they operate with 72 MHz frequency.
ARM Cortex-M3 core is the microcontroller *CPU* and is one of the most significant parts of the microcontroller. This core is the last version of ARM processors which is applied for embedded system. It has well specifications such as 72 MHz maximum frequency, 90DIMPS (Distributed Integrated message Processing System) with 1.25 DIMPS/MHz, performance at zero state memory access, Single-cycle multiplication and hardware division, Nested interrupt controller (maskable interrupt 43 channels, Interrupt processing), Low-power consumption, Low-price, Low-gate count, etc.

Memory system of this microcontroller consists of two parts which they are Flash memory and SRAM (Static Random Access Memory) memory. Flash memory is for storing data and program and its capacity is up to 128 Kbytes. SRAM memory is to read/write at CPU with zero wait state in order to store data for processing by USB and its capacity is up to 20 Kbytes.

Nested Vect It Ctrl (NVIC) stands for Nested Vector Interrupt Controller. It can be used to control up to 43 maskable interrupt channels (maskable interrupt is a special interrupt which could be enabled/disabled or manage by the programmer) and, it has 16 programmable priority levels. It is used to set IRQ (Interrupt Request) channel priorities.

Ext. ITs (EXTI) stands for External Interrupt/Event Controller; it has nineteen edge detector lines in order to create interrupt/event requests. In order to detect external triggers can be used, can be trigged by rising, falling or both trigger and it is also maskable. It is possible to be connected up to eighty GPIOs to sixteen external interrupt lines to detect external triggers (as interrupts).

Clock system (SYSCLK) consists of different clock sources in the microcontroller as well as: HSI (High Speed Internal) oscillator clock, HSE (High Speed External) oscillator clock, PLL (Phase Locked Loop) clock, and LSI RC (Low Speed Internal Resistor and Capacitor oscillator) and LSE (Low Speed External) Oscillator.

HSI oscillator clock is a High Speed Internal clock signal that is an internal (8 MHz) RC oscillator. It is always applied as a clock system for MCU by default.

HSE is a High Speed External clock signal and it can use 4 to 16 MHz external oscillator in order to produce a very accurate clock signal in order to prepare the clock system.

PLL can generate accurate and stable output signal from a fixed low frequency. It multiplies the HSI RC output or HSE crystal output clock frequency to create different frequencies for the microcontroller processor (CPU) and selected peripherals.

The LSI RC clock is a Low Speed Internal clock signal and its clock frequency is about 40 KHz. The LSI RC clock can be used for low power clock source and it is useful in Stop or Standby mode.

The LSE Oscillator is a Low Speed External crystal, its frequency is 32.768 KHz and prepares Real Time clock. These clock sources provide frequency clock for each part of the microcontroller such as CPU, peripherals, etc.

Power Supply is to bias the microcontroller to prepare sufficient electrical power for different parts of system. It is described which pins must connect to the power supply in the following.

- V_{DD}: This pin prepares power supply for I/Os and Voltage Regulator (internal). V_{DD} should be connected to 2.0 to 3.6 Volts and this voltage is provided externally via V_{DD} pin on the MCU.

- V_{SSA} , V_{DDA}: These pins prepare external analogue power supply for ADC, Reset blocks, RCs and PLL. These pins (VSSA , VDDA) should be connected to 2.0 to 3.6 Voltage.

- V_{BAT}: This pin prepares power supply for RTC (Real Time Clock), Internal Clock oscillator (32 KHz) and Backup registers.

Direct Memory Access (DMA) transfers data from Memory to Memory, Memory to Peripheral and Peripheral to Memory. DMA consists of seven channels (each channel is used by the hardware which it requests DMA) and it can be applied with main peripherals such as ADC (Analogue to Digital Converter), TIMx (General Purpose and Advanced Control Timers), SPI, USART, I²C, etc.

Real Time Clock (RTC) *and Backup registers* prepares a 32 bits programmable counter for this MCU in order to provide a set of continuously running counters. They are applied for preparing a Clock Calendar function, a periodic interrupt and an alarm interrupt. Power is provided by VDD or VBAT pins for the RTC and Backup registers.

General Purpose Timers (TIMx) makes up three standard timers which are able to be synchronized in order to be applied for MCU. Each of these timers consists of 16 bits counter (auto re-loadable), Pulse mode output or PWM (Pulse Width Modulation), and they also have a 16 bit prescaler and four independent channels in order to capture input/output or compare.

Advanced Control Timer (TIM1) is as the same as TIMx if it is configured as a standard 16 bits timer, but it is more complete. It can use four independent channels for Input capture, Output compare, PWM generation, etc. It also can work as a three phase PWM multiplexed on six channels.

Inter-Integrated Circuit (I²C) is a bus interface and can perform in two modes which these modes are slave and master. This microcontroller consists of up to two **I²C** bus interfaces.

Universal Synchronous / Asynchronous Receive Transmitter (USART) is a serial port and there are two USARTs are available. One of them can communicate up to 4.5 Mbit/S and another one is up to 2.25 Mbit/S.

Serial Peripheral Interface (SPI) is a serial port and it is able to communicate at speeds up to 18 Mbit/S. There are two SPIs are available which can use DMA controller.

Controller Area Network (CAN) is a standard bus which it is used to make communication between microcontrollers and devices without using any host (computer). It can transact with speed of 1 Mbit/S.

Universal Serial Bus (USB) is a serial bus which it is used in order to transact data between MCU and PC (host). There are four kinds of USB which they are grouped according their speeds.

- Low Speed: Its rate is 1.5 Mbit/S
- Full Speed: Its rate is 12 Mbit/S
- Hi Speed: Its rate is 480 Mbit/S
- Super Speed: Its rate is 5 Gbit/S

The USB peripheral for this MCU is Full Speed version (12 Mbit/S).

General Purpose Input / Outputs (GPIO) is an interface for the MCU in order to connect it to external devices which it can use as input to read data or use as output to write data, etc. There are available three GPIOs on this MCU (STM32F103 with 80 Pins). They should configured by software and they are significant to consider during working with the MCU. All peripherals of the MCU can connect to the outside or other external devices by GPIOs.

Analogue to Digital Convertor (ADC) converts a continuous input analogue signals to digit values in order to use by digital devices such as MCU, PC, etc. This electronic device receives an analogue voltage or current as an input in order to create discrete values at its output. This microcontroller has two ADCs which their resolution is 12 bits and each one has 16 channels.

13. Micro-controller Configuration

The micro-controller configuration set the frequency clock for the Micro controller's CPU and its peripherals. Configuration selects the type of the microcontroller's oscillator and set the exact frequency for each one of the microcontroller component. It also defines the microcontroller's pins for the peripherals as inputs/outputs, operation modes, etc. It just configures the necessary peripherals which they are needed and selected for the project. In this work, CPU, NVIC, GPIO, and USB should be configured by the program in order to run the microcontroller.

In order to configure the microcontroller, it is necessary to define each pin, port or peripheral which it is plan to apply during the project. For example, if it is decided to use the USB peripheral at project, it must be considered all the USB definition. It should be defined which pins of the microcontroller belongs to the USB, the USB clock must be defined and also the USB peripheral must be activated to apply for the project. All these points must be mentioned at configuration part of the microcontroller.

Configuration part of the microcontroller program is always executed by the program in advanced before any thing. Microcontroller configuration is operated after each microcontroller reset; it means configuration of the microcontroller is rebooted when ever the microcontroller is reset. In the following part, there is more explanation for the MCU configuration.

RCC Configuration

The first step in the configuration prepares the system clock for the microcontroller and it is significant to be first because the system clock turns on the microcontroller processor and also it is necessary to active the peripherals (each peripherals need clock to work). By configuring the RCC, it is possible to run the system clock. In this part it is decided which oscillator could be applied for the microcontroller.

The source clock should be defined for the microcontroller (CPU and all peripherals) in the RCC configuration because this microcontroller consists of different kind of source clocks. It is important to pay attention to the microcontroller clock structure before setting clock sources for the microcontroller components. The microcontroller clock structure is named Clock Tree in the microcontroller STM32 documents.

The High Speed External oscillator is defined as the source clock for the microcontroller (In this project), because it is possible to achieve 48 MHz (used for USB clock frequency) if HSE is used as a PLL clock input. Another reason is that it can generate very accurate clock frequency for MCU. So, the HSE is preferred to generate as source clock frequency for the CPU and each peripheral in the microcontroller.

After this configuration, the proper clock frequency for the CPU and peripherals can be set. Then the CPU can start to work and the peripherals are ready to use. The HSE oscillator is used as a PLL clock input and the output of the PLL is SYSCLK (System Clock Source). This SYSCLK frequency should be adjust for different parts of the microcontroller, because each peripheral works with a special frequency clock.

In this configuration, frequency clock for all these parts PLLCLK (PLL Clock), SYSCLK, PCLK2 (Internal APB2 clock), PCLK1 (Internal APB1 clock), should be set, because they prepare clock frequency for all the peripherals.

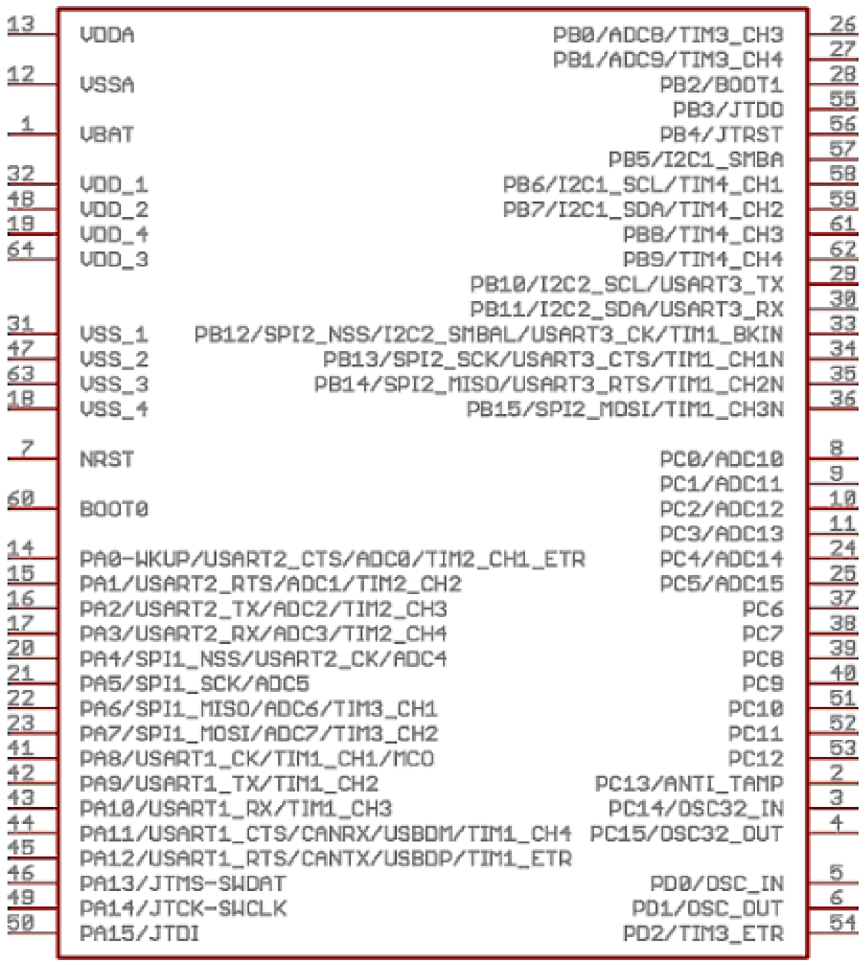
The frequency clock for USB given to be 48Mhz.

The RCC Configuration must be consider as the most important microcontroller configurations (it has the first priority).

GPIO Configuration

The GPIO configuration selects and enables the microcontroller ports or any pins of the microcontroller in order to apply for the microcontroller during the task. This configuration defines inputs/outputs and the status of each port or each pin.

Figure below shows the Pins Configuration of the microcontroller STM32F103. This pins configuration is important in order to configure the microcontroller ports or pins for programming the microcontroller and it should be considered. The number of pins and their specification is mentioned at the figure and it is useful for configuring the microcontroller pins.


13 VDDA
12 VSSA
1 VBAT
32 VDD_1
48 VDD_2
19 VDD_4
64 VDD_3
31 VSS_1
47 VSS_2
63 VSS_3
18 VSS_4
7 NRST
60 BOOT0
14 PA0-WKUP/USART2_CTS/ADC0/TIM2_CH1_ETR
15 PA1/USART2_RTS/ADC1/TIM2_CH2
16 PA2/USART2_TX/ADC2/TIM2_CH3
17 PA3/USART2_RX/ADC3/TIM2_CH4
20 PA4/SPI1_NSS/USART2_CK/ADC4
21 PA5/SPI1_SCK/ADC5
22 PA6/SPI1_MISO/ADC6/TIM3_CH1
23 PA7/SPI1_MOSI/ADC7/TIM3_CH2
41 PA8/USART1_CK/TIM1_CH1/MCO
42 PA9/USART1_TX/TIM1_CH2
43 PA10/USART1_RX/TIM1_CH3
44 PA11/USART1_CTS/CANRX/USBDM/TIM1_CH4
45 PA12/USART1_RTS/CANTX/USBDP/TIM1_ETR
46 PA13/JTMS-SWDAT
49 PA14/JTCK-SWCLK
50 PA15/JTDI
PB0/ADC8/TIM3_CH3 26
PB1/ADC9/TIM3_CH4 27
PB2/BOOT1 28
PB3/JTDO 55
PB4/JTRST 56
PB5/I2C1_SMBA 57
PB6/I2C1_SCL/TIM4_CH1 58
PB7/I2C1_SDA/TIM4_CH2 59
PB8/TIM4_CH3 61
PB9/TIM4_CH4 62
PB10/I2C2_SCL/USART3_TX 29
PB11/I2C2_SDA/USART3_RX 30
PB12/SPI2_NSS/I2C2_SMBAL/USART3_CK/TIM1_BKIN 33
PB13/SPI2_SCK/USART3_CTS/TIM1_CH1N 34
PB14/SPI2_MISO/USART3_RTS/TIM1_CH2N 35
PB15/SPI2_MOSI/TIM1_CH3N 36
PC0/ADC10 8
PC1/ADC11 9
PC2/ADC12 10
PC3/ADC13 11
PC4/ADC14 24
PC5/ADC15 25
PC6 37
PC7 38
PC8 39
PC9 40
PC10 51
PC11 52
PC12 53
PC13/ANTI_TAMP 2
PC14/OSC32_IN 3
PC15/OSC32_OUT 4
PD0/OSC_IN 5
PD1/OSC_OUT 6
PD2/TIM3_ETR 54

The microcontroller peripherals are available from pins of Port A, Port B and Port C. So, it is necessary to active these ports by the program according to the work request and it are obtained at GPIO Configuration. In this configuration, GPIO clock is enabled for all the microcontroller ports in order to enable its ports.

PD2 is used to enable the USB device and mode of this pin should be set as an Output. Program set or reset (1 or 0 voltage) PD2 in order to active the USB device.

PB5 and PB0 are enabled as an output and their mode is set as Output Push Pull. They are used to turn on two LEDs.

PC5,PC6,PC7,PC8 are enabled as output open drain(OD) and PB12,PB13,PB14,PB15 are enabled as input pull up(IPU)

NVIC Configuration
At the NVIC configuration, the interrupt priorities of peripherals are set by the program. This configuration gives the first interrupt priority to the USB peripheral.

USB Configuration
The USB configuration sets and enables the USB clock and all USB status. This configuration defines the USB clock frequency and enables it in order to be used at the project.

The hardware.c is modified for micro-controller configuration in the firmware.

14. STM32USB Preparation

In order to prepare USB communication, the STM32 Performance Stick is used. It has a USB firmware library and software package (it is used to control the STM32F10xxx USB 2.0 full speed device peripheral).

It consists of a Demo USB mouse example. The aim of this firmware library is to provide resources for developing USB application easier and faster in the STM32F10xxx microcontroller family.

The usb_desc.c, usb_desc.h, usb_main.c, usb_main.h, and usb_conf.h should be modified for the USB program according to the project.

The **usb_desc.c** file consists of the USB descriptors for the USB device and it describes the USB device information to the host. This information defines name of the device, version of the USB (USB 2.0), the USB device manufacture, the number of endpoints and endpoint types, etc.

Usb descriptors were explained in the previously.

This source file (usb_desc.c) defines USB Standard Device Descriptors, Configuration Descriptors, Interface Descriptors, Endpoint Descriptors and USB String Descriptors.

The **Device descriptor** represents the device completely and there is only one device descriptor for the USB device and consists of the bLength, bDescriptorType, bcdUSB, bDeviceClass, bDeviceSubclass, bDeviceProtocol, bMaxPacketSize0, id Vendor, id Product, bcdDevice, iManufacturer, iProduct and bNumConfigurations.

bLength: It is the length of the descriptor in byte and is **18 (12 hex)** bytes. It means 18 bytes are required to write the USB device descriptor.

bDescriptorType: It is Device descriptor type and is to describe the type of each part in the USB descriptor (its value is**1h**).

bcdUSB: The binary coded decimal USB states the version of the USB specification. 0100h defines version 1.00; 0110h is for version 1.1 and version 2.00 is specified by 0200h in the program (0200h defines that the USB 2.00 is the USB version for the device). It also mentions that the USB format code is in binary.

bDeviceClass, bDeviceSubclass and bDeviceProtocol: The standard device and interface descriptor consists of field which they are related to the classification: class, subclass and protocol. They applied for the operation systems in order to recognize a class driver for the USB device.

The **bDeviceClass** can be only set at the device level and most class specifications identify their selves at the interface level. The host can use these fields to associate a device or interface to a driver. There is no class defined in the USB in this project, so there are also no subclass and protocol. The valid value for all should be **0x00**.

Then the **bDeviceSubclass** and **bDeviceProtocol** are set **0x00** because of the bDeviceClass.

bMaxPacketSize0: It set the maximum packet size for endpoint0 in the USB device. For high speed devices should be used **0x40 or 64 bytes** (Maximum bytes can be send or receive by the USB buffer is 64 bytes). The host can read this maximum package size for each endpoint from device descriptors.

idVendor, idProduct and bcdDevice: These numbers are defined in device descriptor that they uniquely identify a device to a host. The host can use them to decide what driver should be loaded for the device. For this project, the Vendor ID is **0x0483**, Product ID is **0x1010** and bcdDevice is **0x0000**, because it is not used.

bNumConfigurations: It is to define the number of possible configurations which is supported by the device. The number of configuration is one in this project (0x01), it means device only support one configuration.

The Configuration Descriptor is the second part of the USB descriptors (usb_desc.c). It describes the number of interfaces, the maximum power and how the device is powered. It is possible for the device to have several different configuration descriptors, but most of the devices only have one.

The Configuration Descriptor consists of the bLength, bDescriptorType, wTotalLength, bNumInterfaces, bConfigurationValue, iConfiguration, bmAttributes and MaxPower.

The **bLength** set the size of the configuration descriptor and it is **0x09 bytes (9 bytes)**. It means it is necessary nine bytes to be considered to describe the configuration descriptor for the device.

The **bDescriptorType** defines the type of the descriptor and its type is configuration descriptor.

The **wTotalLength** is the total length in bytes of data returned. After reading the configuration descriptor by the host, it returns the configuration hierarchy that makes of all related interface and endpoint descriptors. The wTotalLength field shows the number of bytes in the hierarchy. The no of returned bytes in this project is the wTotalLength value is **34 (0x22).**

The **bNumInterfaces** states the number of available interfaces for the configuration. There is only one interface available and the bNumInterfaces value is **1 (0x01)** at this project.

The **bConfigurationValue** is applied by the Set_Configuration request in order to select this configuration. It should be 1 or higher and it is set for this device one **(0x01)**.

The **iConfiguration** is an index to a string descriptor which describes the configuration in human readable form. The iConfiguration value is zero if the device does not have string descriptor. It is zero **(0x00)** for the device.

The **bmAttributes** describes power parameters for the configuration which is a bus powered or a self powered device. It is defined bus powered for this device and its value should be **160(0xA0)**.

The **MaxPower** defines how much the maximum power will be drained by the device from the bus. The device is considered bus powered, so the current drawn is 100 mA. Then the MaxPower value is **100 (0x32)**.

The **Interface Descriptors** can be considered as a grouping of endpoints into functional group performing a single feature of the device. It is to states zero or more endpoints for the endpoints descriptors. It contains the information regarding endpoint transfers data type.

The Interface Descriptors describes the bLength, bDescriptorType, bInterfaceNumber, bAlternateSetting, bNumEndpoints, bInterfaceClass, bInterfaceSubClass, bInterfaceProtocol and iInterface.

The **bLength** sets the size of the interface Descriptors for the device and its length is defined nine **(0x09 bytes)**.

bDescriptorType defines the type of the descriptor and it is interface descriptor for this part.

The **bInterfaceNumber** describes the number of interfaces. Its based value is zero in order to introduce at least one interface descriptor (zero only presents one interface device). For more interface descriptor the bInterfaceNumber should increase according to the number of interface devices. This is zero for this device because there is only one USB device to use USB connection.

The **bAlternateSetting** set the alternative interfaces. It is to use when there are more than one interface to communicate via USB. Its value is set zero because there is only one interface needed for this project. If we have two interfaces in this project, then the bInterfaceNumber's value should be set one and the bAlternateSetting also should be set one.

The **bNumEndpoints** defines the number of endpoints which is used for the interface. Its value is set one for this project, because only one endpoint is considered for this device in this project. The endpoint **1 (IN1)** is use to send data to PC.

The bInterfaceClass, bInterfaceSubClass and bInterfaceProtocol specify supported classes. Classes are such as, HID (The Human Interface Device), Communication, Mass Storage, etc. They allow many devices use the same drivers and it is not needed to write specific driver for the device.

The **bInterfaceClass** value is set to **3 (0x03)** which defines HID Interface class.

The **bInterfaceSubClass** value is set to **1 (0x01)** which defines that the device will also work in BOOT protocol.

bInterfaceProtocol value is set to **1 (0x01)** which defines that the HID device is a keyboard

The **iInterface** is an index of String Descriptor to describe this interface in order to allow having a string descriptor for the interface. This definition does not use in this project, so its value is zero.

The **Endpoint Descriptors** describes the endpoints and each endpoint must be defined instead of endpoints zero (endpoint0). Endpoint0 is always described by default and it is supported by every devices. Therefore it never has a descriptor. The host system learns about endpoints by studying the configuration descriptor which each endpoint belongs to it. At this program, only one endpoints is defined, so it should be described at the endpoint descriptors file.

The Endpoint Descriptor 1 is described and it includes the bLength, bDescriptorType, bEndpointAddress, bmAttributes, wMaxPacketSize and bInterval.

The **bLength** sets the size of the Endpoint Descriptors in byte and its size is seven **(0x07)**. This size is equal for any endpoint descriptors.

The **bDescriptorType** defines the type of the descriptor and the type of descriptor is Endpoint Descriptors.

The **bEndpointAddress** defines the endpoints number which it is used. The Endpoint 1 is set and its address is **0x81**. (Bits 0 to 3 define endpoint number and bits 4 to 6 are zero as reserve and bit 7 should be zero or one. If it is one, it means endpoints is for IN and if it is zero, it means endpoint is for OUT).

The **bmAttributes** is to define the type of the transfer type. Four types of data transfer are available which they are selectable. They are Control (00b), Isochronous (01b), Bulk (10b) and **Interrupt (11b)**. For this project, transfer type is set Interrupt and it should be defined by 11 binary or 0×03 hex.

The **wMaxPacketSize** defines the Maximum Packet size for the endpoints which it is possible to transfer in a transaction. It is defined as **8 bytes (0x08)** for sending an 8 byte keyboard IN report.

The **bInterval** is set to specify the maximum polling interval of certain transfer. It is defined as **64ms (0x40)** for polling the Endpoint0 for keydata to be sent to Host.

The USB **String Descriptors** includes descriptive text and it is to define the USB descriptors. It prepares human readable information and it is optional. It explains the language, manufacturer, product and serial number for the device.

The **bLength** defines the size of the descriptor in bytes.

The **bDescriptorType** describes the type of the descriptor.

The manufacturer, product and serial number contain the bLength, bDescriptorType and bstring.

The **usb_desc.c** file also contains the Report Descriptor which defines the report to be sent to Host from the device. Report Descriptor has been explained in the section ?

The **usb_desc.h** file is a header file or include file for the usb_desc.c file. It is needed to define setting for the constants which they are used in the usb_desc.c. This file should be prepared according to the setting in the usb_desc.c.

Now the microcontroller STM32F10 can use the USB peripheral to plug in to the host and the host system can recognize the USB in order to communicate with the microcontroller. This description regarding the USB communication is general and it does not matter which kind of microcontroller is chosen. It is also possible to set the other microcontrollers which the USB peripheral is available for them. It is possible to apply above explanation to prepare USB communication for other version of micro-controllers.

List of values of USB descriptors

DEVICE DESCRIPTOR

bLength	**= 0x12u,**
bDescriptorType	**= 0x01u,**
bcdUSB	**= 0x00u, 0x02u,**
bDeviceClass	**= 0x00u,**
bDeviceSubClass	**= 0x00u,**
bDeviceProtocol	**= 0x00u,**
bMaxPacketSize0	**= 0x08u,**
idVendor	**= 0x83u, 0x04**
idProduct	**= 0x10u, 0x57**
bcdDevice	**= 0x00u, 0x02u,**
iManufacturer	**= 1**
iProduct	**= 2**
iSerialNumber	**= 3**
bNumConfigurations	**= 0x01u**

CONFIG DESCRIPTOR

bLength	**= 0x09u,**
bDescriptorType	**= 0x02u,**
wtotallength	**= 0x22u, 0x00u,**
bNumInterfaces	**= 0x01u,**
bConfiguration Value	**= 0x01u,**
iConfiguration	**= 0x00u,**
bmAttributes	**= 0xA0u,**
bmaxPower	**= 0x32u,**

INTERFACE DESCRIPTOR

bLength	**= 0x09u,**
bDescriptorType	**= 0x04u,**
bInterfaceNumber	**= 0x00u,**
bAlternateSetting	**= 0x00u,**
bNumEndpoints	**= 0x01u,**
bInterfaceClass	**= 0x03u,**
bInterfaceSubClass	**= 0x01u,**
bInterfaceProtocol	**= 0x01u,**
iInterface	**= 0x00u,**

ENDPOINT DESCRIPTOR

bLength	**= 0x07u**
bDescriptorType	**= 0x05u,**
bendpointAddress	**= 0x81u,**
bmAttributes	**= 0x03u,**
wmaxPacketSize	**= 0x08u,**
binterval	**= 0x40u**

15. Keypad Interfacing and sending data

Once the microcontroller STM32F103 is enumerated, it can used as a HID keyboard device.
Now the aim is to attach a keypad matrix to the GPIO's of the micro-controller and to scan the matrix for checking if any key is pressed, and finally to send the key press data as IN report to host.

The above process is explained bit more in detail.

A 4x4 keypad matrix is connected to the micro-controller GPIO pins. The keypad contains 8 wires that is to be connected to the micro-controller. The 4 wires are for rows and other 4 for columns.

The column pins should be configured as input with pull up (IPU) since these are connected to V_{cc} through pull-up resistors.

The row pins should be configured as outputs with open drain mode (OD) to connect them to ground.

Now each row was grounded one by one and state of all the column pins was read each time. If any key is pressed in that row, it will read '0'.

Pseudo code:

```
PC5,PC6,PC7,PC8 pins are configured as OD.
PB12,PB13,PB14,PB15 pins are configured as IPU.
// Set outputs high-level

GPIO_SetBits(GPIOC, GPIO_Pin_5);
GPIO_SetBits(GPIOC, GPIO_Pin_6);
GPIO_SetBits(GPIOC, GPIO_Pin_7);
GPIO_SetBits(GPIOC, GPIO_Pin_8);

// Bring each output scan line low, wait briefly, then read then read the input stated to identify key
pressed.

GPIO_ResetBits(GPIOC, GPIO_Pin_5); //Drive pin to gnd
delay(10);
if (GPIO_ReadInputBit(GPIOB,GPIO_Pin_12) == 0) // Check if column1 pin is pressed
   key=0x04; // send scan code assigned to the button
if (GPIO_ReadInputBit(GPIOB,GPIO_Pin_13) == 0) // Check if colum2 pin is pressed
   key=0x05; // send scan code assigned to the button
//continue to check all colums.

GPIO_SetBits(GPIOC, GPIO_Pin_5); // Release row1 HIGH

// Repeat the above steps for PC6,PC7,PC8
```

For 'SHIFT+ key' to be used row4 col4 button was assigned as SHIFT button and if that key is pressed and is held a 'flag' variable is made HIGH.

The flag is HIGH and if a button is pressed it means that the SHIFT key is pressed and different scan code than scan code if only the button (flag=0) is pressed is sent.

Example:

```
 if(GPIO_ReadInputDataBit(GPIOB,GPIO_Pin_12)==0 & flag)
  // check if shift key and row1 col1 key    is pressed
 {
    delay(10);  //switch debounce delay
       if(GPIO_ReadInputDataBit(GPIOB,GPIO_Pin_12)==0 & flag) //check again after delay
          {
            return A;          //if true send capital 'A'
           }
  }
```

As previously explained the key press data is sent to the host as an 8 byte IN report. The key press data is stored in a variable 'key' and modifier data is is stored in the variable 'mod'.

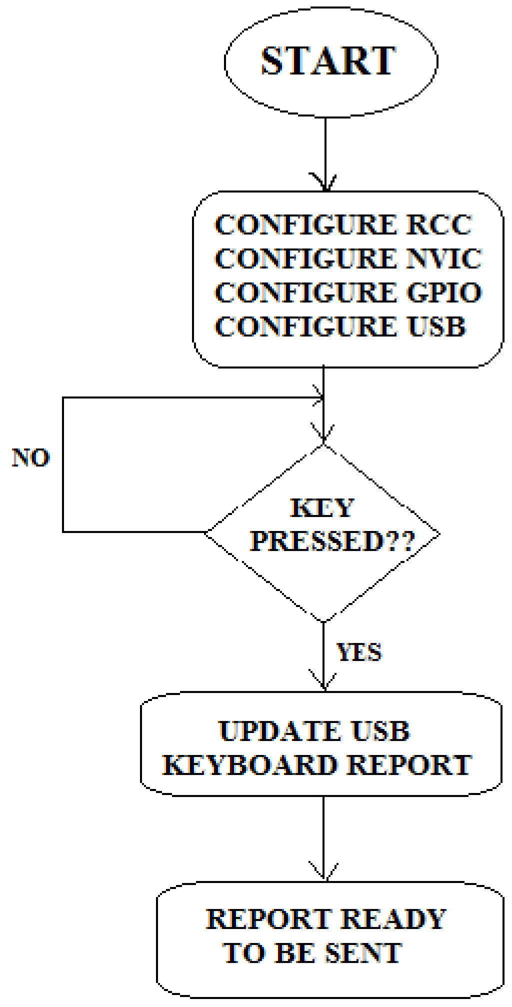
```
/* Key press data is stored in an array according to USB keyboard input report format */

             hid_report[0]=mod;
             hid_report[1]=0x00;
             hid_report[2]=key;
             hid_report[3]=0x00;
             hid_report[4]=0x00;
             hid_report[5]=0x00;
             hid_report[6]=0x00;
             hid_report[7]=0x00;
```

Once we know which key is pressed it should be sent to the host through USB port.
To do this ENDPOINT1 must be set and the hid_report data should be sent to it. The host polls for data in the ENDPONT1 buffer every 64ms.

The library function used to do this given below.

```
  /*copy key press data info in ENDP1 Tx Packet Memory Area*/
     UserToPMABufferCopy(hid_report, GetEPTxAddr(ENDP1), 8);

/* enable endpoint for transmission */
     SetEPTxValid(ENDP1);
```

The C program flow for developing a USB Keyboard is given in the form of a flowchart below.

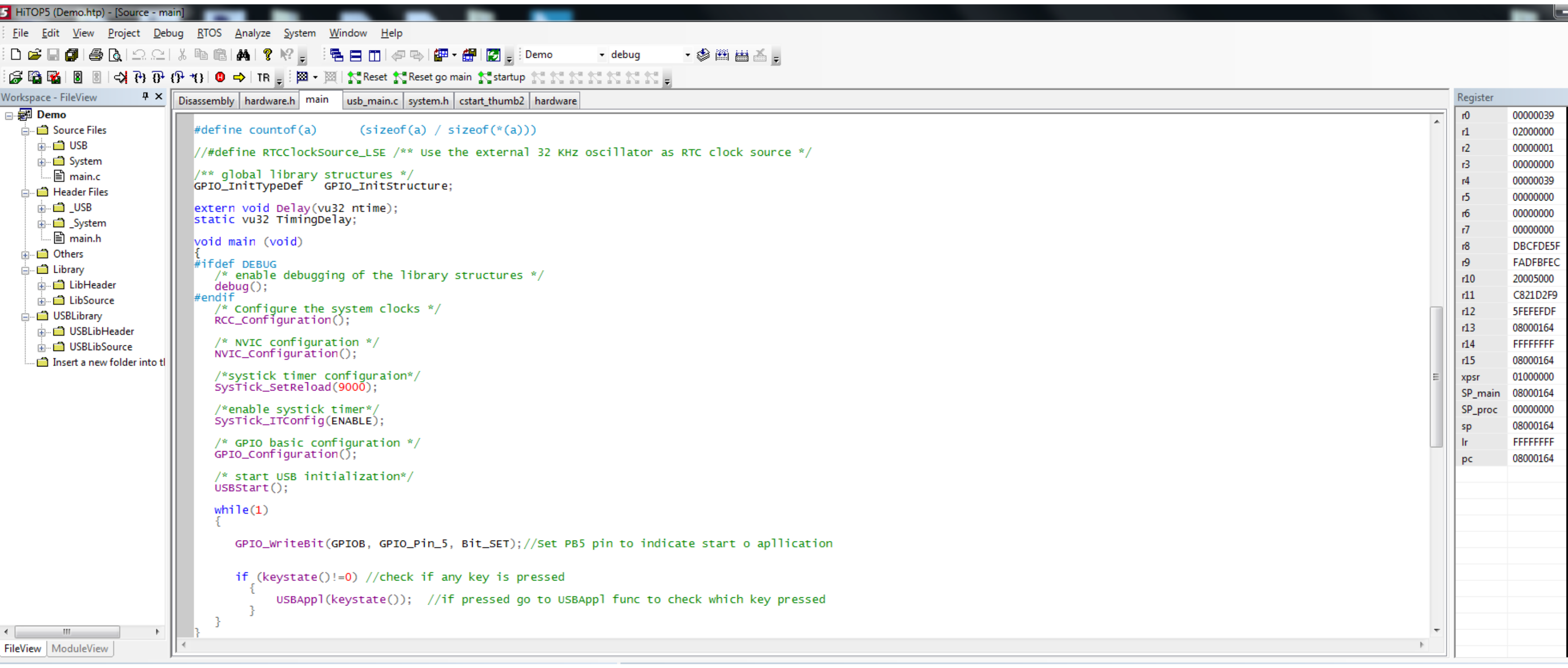


The steps followed are:

1. In the firmware code the necessary initial configurations required for the application is written.
2. The USB descriptors are written accordingly for the STM32 microcontroller to act as USB keyboard.
3. Necessary code for Keypad interfacing to the micro-controller is written.
4. Program is compiled and loaded into the micro-controller.
5. When the program is made to run micro-controller keeps checking if any key is pressed.
6. If any key is pressed USB keyboard report is updated and is sent to the ENDPT1.
7. The host keeps polling the ENDPT1 for new data, and if any present it is sent to the host.

16. TESTING THE PROJECT

HiTop IDE was used to develop the firmware for USB keyboard. The Demo USB mouse example was used as a base for writing the code for USB keyboard.

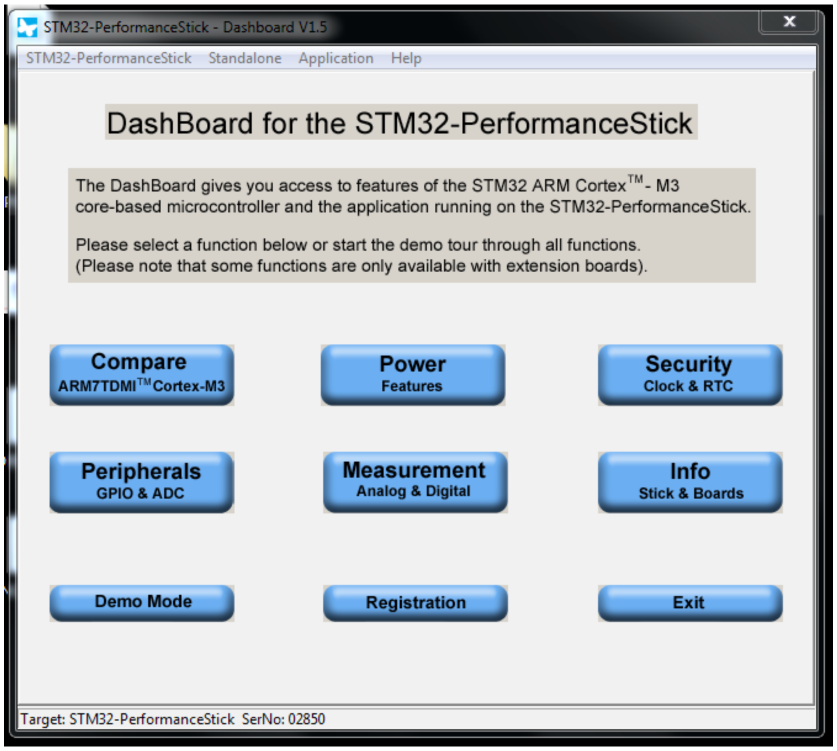


```
#define countof(a)      (sizeof(a) / sizeof(*(a)))

//#define RTCClockSource_LSE /** Use the external 32 KHz oscillator as RTC clock source */

/** global library structures */
GPIO_InitTypeDef   GPIO_InitStructure;

extern void Delay(vu32 ntime);
static vu32 TimingDelay;

void main (void)
{
#ifdef DEBUG
   /* enable debugging of the library structures */
   debug();
#endif
   /* Configure the system clocks */
   RCC_Configuration();

   /* NVIC configuration */
   NVIC_Configuration();

   /*systick timer configuraion*/
   SysTick_SetReload(9000);

   /*enable systick timer*/
   SysTick_ITConfig(ENABLE);

   /* GPIO basic configuration */
   GPIO_Configuration();

   /* start USB initialization*/
   USBStart();

   while(1)
   {

      GPIO_WriteBit(GPIOB, GPIO_Pin_5, Bit_SET);//Set PB5 pin to indicate start o apllication


      if (keystate()!=0) //check if any key is pressed
         {
            USBAppl(keystate());  //if pressed go to USBAppl func to check which key pressed
         }
   }
}
```

r0	00000039
r1	02000000
r2	00000001
r3	00000000
r4	00000039
r5	00000000
r6	00000000
r7	00000000
r8	DBCFDE5F
r9	FADFBFEC
r10	20005000
r11	C821D2F9
r12	5FEFEFDF
r13	08000164
r14	FFFFFFFF
r15	08000164
xpsr	01000000
SP_main	08000164
SP_proc	00000000
sp	08000164
lr	FFFFFFFF
pc	08000164

The demo program was modified for the STM32 Performance Stick to be behave as a USB HID keyboard. After necessary changes the STM32 Performance Stick was enumerated and was behaving as USB keyboard. The STM32 was seen at the PC as an USB HID (Human Device Interface) keyboard device.

For the Demo program given with the Stick, a GUI was used control the various operations such as enabling the USB application to start its application



DashBoard for the STM32-PerformanceStick

The DashBoard gives you access to features of the STM32 ARM Cortex™- M3 core-based microcontroller and the application running on the STM32-PerformanceStick.

Please select a function below or start the demo tour through all functions.
(Please note that some functions are only available with extension boards).

Target: STM32-PerformanceStick SerNo: 02850

To remove the control through GUI, codes related to GUI control were removed and necessary changes were made, and the program loaded into the micro-controller started to run without any external control.

A 4x4 matrix keypad was interfaced with the micro controller and necessary program was written as explained in the previous section.

The keys were working according to the program, but switch de-bouncing problem was found. To solve this problem, after checking for a key press a small delay was given and that key was checked again.

Now all the keys were working correctly. All small letter and number data were sent properly. For SHIFT key to work necessary code was written as explained previously and checked for the same.

CAPS, NUM Lock keys and other functional keys were verified.

The code was optimized to keep only necessary files required for USB keyboard and others were removed from the demo program.

17. Further development

Micro-controller suggestions:

Micro-controller	**STM32F102C8**	**STM32F102CB**	**STM32F103C8**	**STM32F103CB**	**STM32F103RB (used in demo kit)**
Price(1 nos)	Rs. 333.19	Rs. 411.23	Rs. 411.23	Rs 436.40	Rs. 460
No of I/O's	37				51
RAM	10 KB	16 KB	20 KB	20 KB	20 KB
Pin	48				64
Flash memory	64 KB	128 KB	64 KB	128 KB	128 KB
CPU Speed	48 MHz	48 MHz	72 MHz	72 MHz	72 MHz

Current program size:
RAM Used: 4.36 KB
Flash Memory used: 10.28 KB
hex file size: 27 KB
No of I/O pins reqd: 25-27

Components Required:

COMPONENT	**SPECS**	**PRICE**
External crystal	8MHz, 18pF load capacitance	Rs 9.31
USB connector	USB B type	Rs 68
ST322CTR IC	USART Transceiver IC	Rs 38.51
Resistors	(1.5Kx1), (27x2), (10Kx2)	
Capacitors	(100nFx5), 4.7microF, (29pFx2)	
Diodes	3.6V breakdown voltage (2 nos)	

Loading a program to the STM32 micro-controller

The bootloader is stored in the internal boot ROM memory (system memory) of STM32 devices. It is programmed by ST during production. Its main task is to download the application program to the internal Flash memory through one of the available serial peripherals (USART, CAN, USB, I2C, SPI, etc.)

In STM32F10xx series bootloader supports only USART serial interface to download code into internal Flash memory.

To enable this BOOT1 pin in the micro-controller should be set to '0' and BOOT0 pin should be set to '1'. Once this pins are configured, after reset execution starts at System memory where built in bootloader resides.

A Flash Loader Demonstrator software is used on the PC to load the hex or bin file to micro-controller memory.

When programming is done BOOT0 pin is returned to ground in order to execute programs from main Flash memory after next reset.

In the circuit side an USART Transceiver IC is used generate RS232 voltage levels from 5V power supply. USART_1tx pin and USART_1rx pins from the micro-controller are connected to T1In and R1Out pins of transceiver IC respectively. The T1Out and R1In pins of transceiver is connected to DB9 connector through which it is communicated to PC.