USB KEYBOARD PROTOCOL

1. IN Token Packet

- Sent by the **host** (your microcontroller).
- Requests data **from** the device (keyboard) to the host.
- Used for reading keystrokes from the keyboard.
- If the keyboard has data, it responds with a **DATA packet**.
- If no data is available, it responds with a **NAK** (Not Acknowledged) to indicate the host should try again later.

2. OUT Token Packet

- Sent by the **host**.
- Indicates that the host is about to send data **to** the device.
- Followed by a **DATA packet** containing the actual data.
- The device acknowledges receipt with an **ACK** if successful.
- Used for sending commands (e.g., controlling LED indicators like Caps Lock).

3. SETUP Token Packet

- Sent by the **host** at the beginning of a **control transfer**.
- Used for configuring the device (e.g., setting addresses, requesting descriptors).
- Always followed by a **DATA0 packet** containing the request.
- The device responds with a **status stage** after processing.

4. SOF (Start of Frame) Token Packet

- Sent by the **host** every **1ms in Full-Speed (USB 1.1, 12 Mbps)** and **125µs in High-Speed (USB 2.0, 480 Mbps)**.
- Used to keep USB devices synchronized.
- Not usually relevant for basic keyboard communication unless dealing with isochronous data (which keyboards don't use).

How These Work in Keyboard Communication

1. **Enumeration (Setup Stage)**
    - The host sends a **SETUP** token to request descriptors (e.g., Device, Configuration, HID Report).
    - The keyboard responds with descriptor information.
2. **Polling for Keystrokes**
    - The host repeatedly sends **IN** tokens.
    - The keyboard responds with a **DATA** packet if a key is pressed.
    - If no key is pressed, the keyboard responds with **NAK**.
3. **Sending Commands**
    - The host sends an **OUT** token followed by a **DATA** packet (e.g., enabling Num Lock).
    - The keyboard acknowledges with an **ACK**.

HOW TO FIND OUT USB CLOCK FREQUENCY

If your microcontroller has a USB host controller (like STM32, ESP32-S3, or ATmega32U4 in host mode), the hardware should automatically detect the speed. However, if you are handling USB manually, follow these steps:

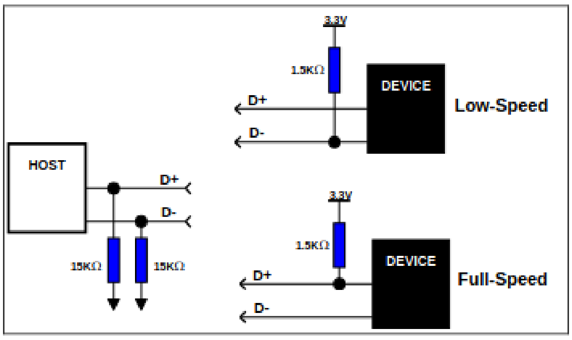
1. **Monitor D+ and D- during device connection**
    - If **D+ is HIGH**, assume **Full-Speed (12 Mbps)**.
    - If **D- is HIGH**, assume **Low-Speed (1.5 Mbps)**.
2. **Perform a USB Reset**
    - Drive **both D+ and D- LOW** for at least **10ms**.
    - Release the lines and observe which one pulls up.
3. **Request the Device Descriptor**
    - Send a **Get Descriptor (Device)** request using a **SETUP token**.
    - Read the `bcdUSB` field to confirm the USB version.

When a USB device is connected to a host, the speed of the device needs to be detected. This is done with pull-up resistors on the D+ or D- line. A 1.5-kΩ pull-up on the D+ line indicates that the attached device is a Full-Speed device. A 1.5-kΩ pull-up resistor on the D- line indicates the attached device is a Low-Speed device. This can be seen in Figure 17.

High-Speed devices start as Full-Speed devices, so they have a 1.5-kΩ pull-up on the D+ line. When the device is connected, it emits a sequence of J-States and K-States during the reset phase of enumeration. If the hub supports High-Speed, then the pull-up resistor is removed.

The pull-up resistor is essential to USB enumeration. Without the pull-up resistor, USB assumes that there is nothing attached to the bus. Some devices require an external pull-up resistor on the D+/D- line. PSoC, however, implements the required pull-up resistor internal to the device, which eliminates the need for this external component.

Figure 17. USB Speed Detection



PORTD0 == D-   PORTD1 == D+

	D+	D-	*Page 145*
Differential 0	Lo	Hi	
Differential 1	Hi	Lo	
Single-Ended 0	Lo	Lo	
Single-Ended 1	Hi	Hi	
J State		Differential 0 (Low Speed)	
K State		Differential 1 (Low Speed)	
Bit 1		No change in level	
Bit 0		Change in level	

1. **(Start of every packet) SYNC PATTERN:** KJ KJ KJ KK
2. **PID:** 8bits , PID0 PID1 PID2 PID3 NOTP0 NOTP1 NOTP2 NOTP3

PID Type	**PID Name**	**PID[3:0]**	*Page 196*
Token	OUT	0001	
	IN	1001	
	SETUP	1101	
Data	DATA0	0011	
	DATA1	1011	
H/S	ACK	0010	
	NAK	1010	

3. **ADDRESS Field (IN, OUT, SETUP):** Addr[0:6]
4. **ENDPOINT Field:** EP[0:3]   0000 ???
5. **DATA Field:** 0 – 1024 bytes   B0 B1 B2 B3 B4 B5 B6 B7 B’0 …
6. **CRC:** Το CRC υπολογιζεται ΧΩΡΙΣ το bit stuffing
   Generator polynomial for Token Packets: G(X) = X^5+ X^2 + 1 = 00101
   -//- for Data Packets: G(X) = X^16 + X^15 + X^2 + 1 = 1000 0000 0000 0101
   Το CRC δεν εχει να κανει με τα PIDS, τα PIDS εχουν δικο τους error check (4 comp bits)
7. **EOP:** SE0 SE0 J

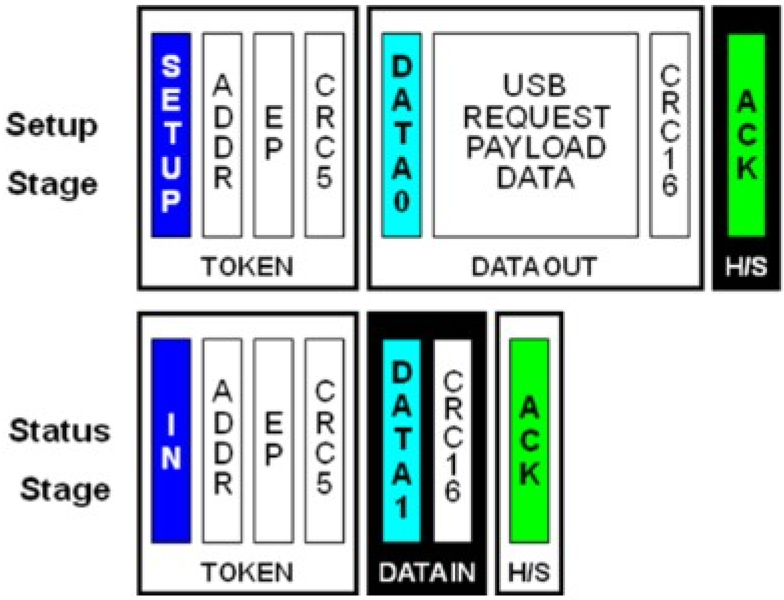
TOKEN PACKET: SYNC PID[8] ADDR[7] EP[4] CRC[5]

DATA PACKET: SYNC PID[8] DATA[0-8192] CRC[16]

HANDSHAKE PACKET: SYNC PID[8] **EOP**

SETUP ADDRESS TRANSACTION

Figure 39. Control No Data Transaction



TOKEN SETUP PACKET: SYNC PID[8] ADDR[7] EP[4] CRC[5]

SYNC	PID	ADDR	EP	CRC	EOP
KJ KJ KJ KK	1011 0100 KJJJ KKJK	0000 000 JKJK JKJ	0000 KJKJ	01000 KKJKJ	SE0 SE0 J

DATA REQUEST PACKET: SYNC DATA0 PAYLOAD CRC[16] EOP

set address

SYNC	PID	RequestType	Request (SetAdd)	Value	Index	Length	Data	CRC	EOP
KJ KJ KJ KK	1100 0011	0000 0000	1010 0000	10000000 00000000	00000000 00000000	00000000 00000000	-	11010111 10100100	SE0 SE0 J
	KKJK JKKK	JKJK JKJK	KJJK JKJK	KJKJKJKJ KJKJKJKJ	KJKJKJKJ KJKJKJKJ	KJKJKJKJ KJKJKJKJ		JJKKJJJJ JKKJKKJK	

ACK **DDRD = 0x00 !!!!!!**

SYNC	PID	EOP
KJ KJ KJ KK	0100 1011 JJKJ JKKK	SE0 SE0 J

TOKEN IN PACKET:

SYNC	PID	ADDR	EP	CRC	EOP
				01000	
KJ KJ KJ KK	1001 0110	0000000	0000	5bits	SE0 SE0 J

EMPTY DATA PACKET:

SYNC	PID (DATA 1)	Data	CRC	EOP
			0	
KJ KJ KJ KK	1101 0010	-	16bits	SE0 SE0 J

ACK

SYNC	PID	EOP
KJ KJ KJ KK	0100 1011	SE0 SE0 J

GET CONFIGURATION DESCRIPTOR
SETUP TOKEN SYNC SETUP ADDR EP CRC EOP
DATA PACKET SYNC DATA0/1 PAYLOAD(request) CRC EOP
ACK

IN TOKEN SYNC IN ADDR EP CRC EOP
DATA PACKET SYNC DATA1/0 PAYLOAD CRC EOP
ACK

OUT TOKEN
EMPTY DATA PACKET
ACK

IN TRANSACTION:

(Host) IN token → (Device) Data/NAK → (Host) ACK/ No response