



# Yet Another Firmware Based Soft USB Device

Zach Lee

[wiki.geniekits.com](http://wiki.geniekits.com) & [github.com/GenieKits](https://github.com/GenieKits)

## Background

It was about 2006. I found a wondrous project on the Internet. It was called “PowerSwitch”, which provides 8 channels of parallel output that intends to be switches. Now it is hosted at <https://www.obdev.at/products/vusb/powerswitch.html>. These switches could be controlled through an USB host but the MCU (ATMEL 90S2313) doesn't have any USB peripheral. Two GPIO pins are used to capture the signals of USB D+/D- and all signals are decoded by the firmware in realtime. Of course, it is a low speed USB device.

The high speed of AT90S2313 is the key point of this project. The MCU runs at 12MIPS when it is driven by a 12MHz crystal. Every eight instructions corresponds to one bit in the USB low speed (1.5Mbps) bit stream. The level of D+/D- could be sampled accurately and decoded before the next bit coming. The firmware just does NRZI decoding and BIT UNSTUFF because 12MIPS is not fast enough to do the CRC verification. When the bit stream is sent to the host, the CRC16 code will be calculated and attached to the bit stream. A handshake NAK is sent to the host repeatedly until the CRC16 code was ready.

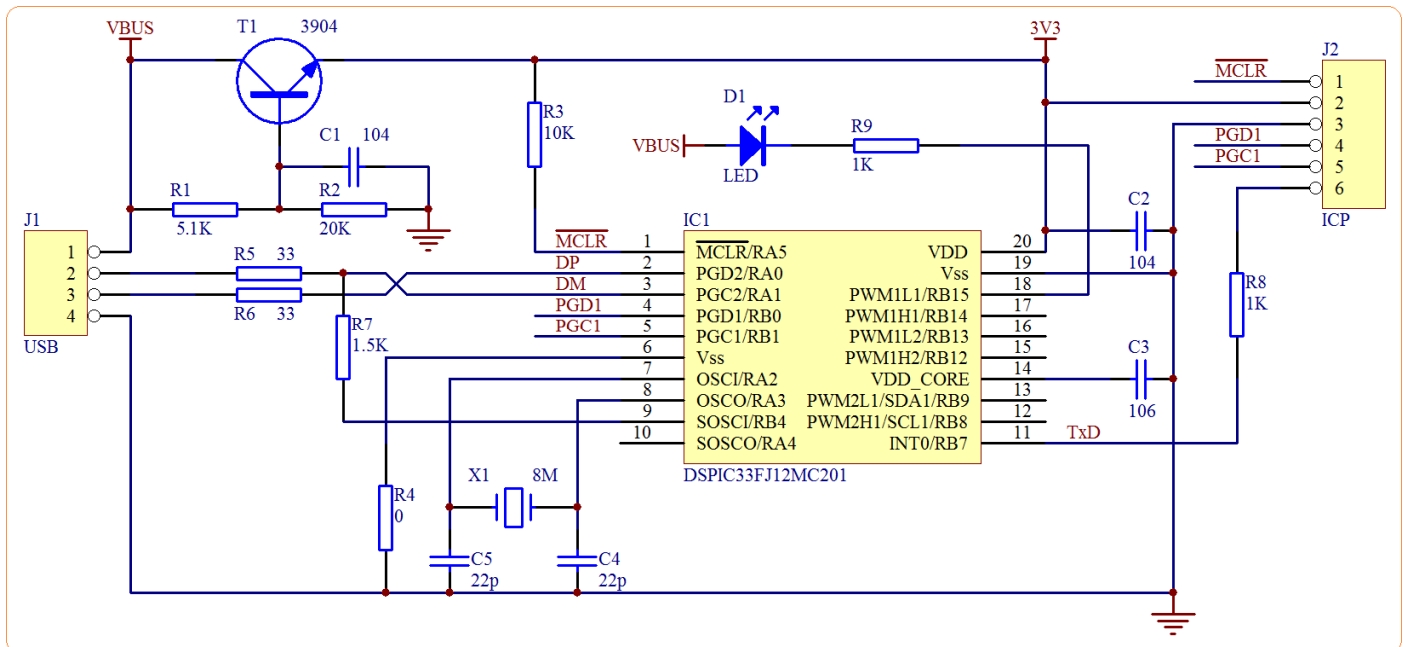
It is a really smart design. There is an application note (AVR309) published by ATMEL Co. in which the technique details are written down. The kernel of this project is coded with assembly language. Unfortunately, I couldn't analyze the complete code of PowerSwitch because I wasn't an expert to AT90xx chips at that time. So far PowerSwitch has been evolved into an independent project which is called "V-USB" and hosted at <https://www.obdev.at/products/vusb/index.html>. It has been ported to "ATMEGA" chips and there are some practical product derived from this project, for example, USBasp. However, it has not been implemented on other chips except ATMEL's.<sup>①</sup>

Years ago I designed a few USB devices. Because the chips I used have USB port, I have never concerned about the low level protocol of USB. In this project I am planning to implement a new firmware based SOFT USB device which is similar as "V-USB". It will bring me better comprehension to the low level protocol of USB. I chose PIC24F/dsPIC33 series chip which is produced by Microchip and coded with C and Assembly language. If you are not an expert to Microchip's MCU, this project might bring great confusion to you. In another words, this project will bring you a great chance to study Microchip's MCU in depth.

---

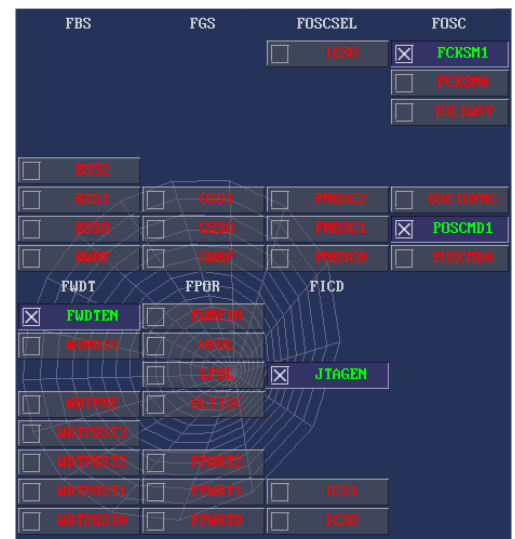
<sup>①</sup> Recently I found a similar project which is based on STM32F030. It is hosted at <https://github.com/ads830e/stm32f030-vusb>.

## The Circuit



The major controller IC1 is dsPIC33FJ12MC201. This chip provides small SSOP package with 20 pins and 40MIPS maximum running speed. I use an 8MHz crystal and generate 30MHz (Fpllout = 30MHz) clock via PLL, driving the CPU running at 15MIPS. Every 10 machine cycles corresponds to one bit in the USB low speed (1.5Mbps) bit stream. I use two pins (RA0/RA1) of GPIO-A to capture the level of USB D+/D-. Because the "Change Notification" interrupt can be triggered by RA0 or RA1 directly, we don't have to use the INT0 pin. We MUST connect USB D+/D- to two pins within one GPIO group. It's not allowed to connect D+ to RAX and D- to RBX. The pull-up resistor R7 on USB D- is NOT connected to 3.3v directly. I use RB4 to control this pull-up resistor. There is a LED connected to RB15. I will send some data through USB port to control this LED. MCLR/PGC1/PGD1 are used as an ICSP port. We can use some burner, such as PICKit3, to program the chip through it. RB7 is connected to the PIN6 of ICSP connector J2. It is used to send some debugging messages when we need to. The PIN6 of ICSP port has been used as the LVP signal of PICKit3. Because I have never used PICKit3, I don't know if this connection (RB7 to PIN6 of ICSP) will make PICKit3 fail or not. There is a NPN transistor T1 which is used as a 3.3V power regulator. On my experimental board it is replaced by an AMS1117-33. I think it will work if you try to connect a red LED at VBUS in series instead of transistor T1, reducing the +5V to 3.2V. The resistor R4 could be replaced by a solder bridge. If you want to use PIC24F16KA101 to instead dsPIC33FJ12MC201, you can just omit R4 and C3 and replace X1 with a 30MHz crystal.

Fuse definitions are not included in the source code. You have to set "FCKSM1/POSCMD1/FWDTEN/JTAGEN" into "Programmed (0)". All other fuses are "Unprogrammed (1)". If you use PIC24F16KA101 to instead of dsPIC33, another four bits "FNOSC2/FNOSC0/POSCMD0/FWDTEN" need to be set to "Programmed (0)".



## Coding Environment

This project is implemented and tested on WINDOWS platform. I use the XC16 compiler suit version 1.25 which is developed by Microchip. Unless the compiler I don't use any integrated development environment (MPLAB IDE) and debug probe (ICD4 or PICKit). I haven't bought any commercial license as well. Without commercial license the GCC compiler only supports -O1 level optimization. It's enough for this project. You can check out this hyperlink listed below to download XC16 compiler suit. It will work if you want to try another version higher than 1.25.

<https://www.microchip.com/development-tools/pic-and-dspic-downloads-archive>

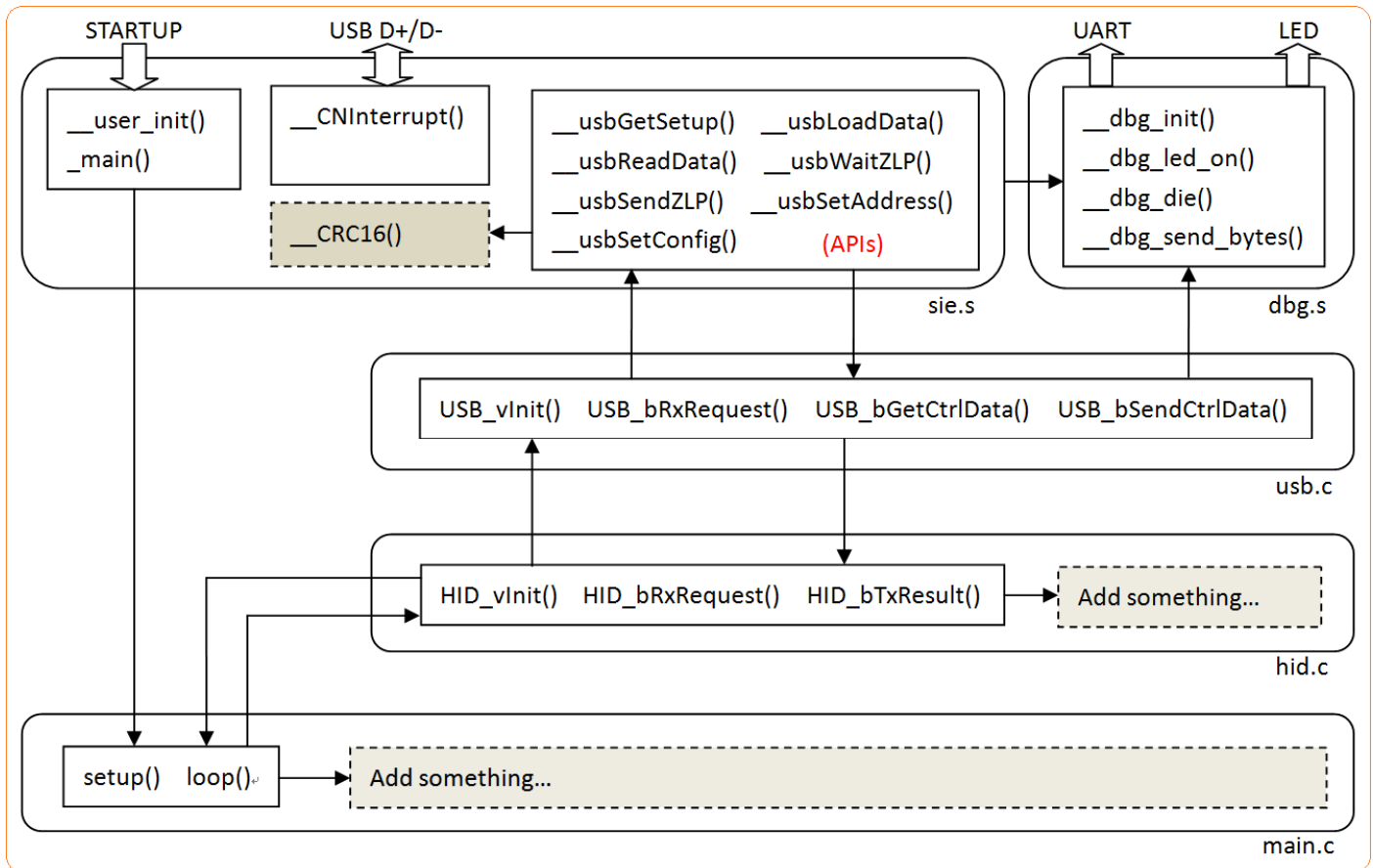
I use a BAT file (usb.bat) to compile the source codes:

```
xc16-gcc -mcpu=33FJ12MC201 -O1 main.c hid.c usb.c sie.s dbg.s -o main.elf -T p33FJ12MC201.gld
-Wl,--defsym,__has_user_init=1,-Map=main.map
xc16-bin2hex main.elf
xc16-objdump -D main.elf >main.txt
pause
```

As you can see I pass a parameter to the linker, -Wl,--defsym,\_\_has\_user\_init=1, then the function named "\_\_user\_init" in the source file "sie.s" will be invoked by the C startup code. In the installation directory of XC16 you can find a source-code cab which is named "libpic30.zip". In this cab there is a source file named "crt0\_standard.s" to be controlled by the symbol "\_\_has\_user\_init".

The program running on the host for testing are coded and compiled with Microsoft Visual Studio 2008. I use a self-made firmware burner to instead of PICKit3 or other debug probe. I'll create some documents for it in the future.

## Structure of the source files



There are five source files in this project. Two of them are assembly source codes (`sie.s/dbg.s`). "`dbg.s`" is used for debugging, sending some messages through the UART port or showing states through the LED. It is NOT a necessary module in this project. Others are C source files. "`usb.c`" is used to handle some standard requests of USB specification, such as "GET\_DESCRIPTOR", etc. "`hid.c`" is used to handle requests (SET/GET REPORT) of HID specification. Initialization code and main task loop are included in "`main.c`". The "main entry" of C language is defined in "`sie.s`". There are only two functions in "`main.c`", `setup()` and `loop()`, just like the source code style of "arduino".

"`sie.s`" is the kernel code of this project. The "Change Notification" interrupt service routine and USB low level protocol processing code are inside this file. There are also some global variables and buffers in this file. A group of API functions are defined in this module for "`usb.c`". It is recommended and easy to invoke these API functions instead of accessing the global variables.

## Details of source codes

### References and online resource

DS52106A : MPLAB\_XC16\_Asm\_Link\_Users\_Guide.pdf

DS50002071E : MPLAB\_XC16\_C\_Compiler\_Users\_Guide.pdf

These two docs are included in XC16 compiler suit. You can find them in the installation directory.

DS70265E : dsPIC33FJ12MC201/202 Data Sheet

<http://ww1.microchip.com/downloads/en/devicedoc/70265e.pdf>

DS39927C : PIC24F16KA101/102 Data Sheet

<http://ww1.microchip.com/downloads/en/DeviceDoc/39927c.pdf>

DS70193C : dsPIC33F/PIC24H Family Reference Manual Section 10 – I/O Ports

<http://ww1.microchip.com/downloads/en/devicedoc/70193c.pdf>

DS70000157G : 16-bit MCU and DSC Programmer's Reference Manual

<http://ww1.microchip.com/downloads/en/DeviceDoc/70000157g.pdf>

USB2.0 Specification : usb\_20\_20190524.zip

<https://usb.org/document-library/usb-20-specification>

Device Class Definition for HID1.11 : hid1\_11.pdf

<https://usb.org/document-library/device-class-definition-hid-111>

HID Usage Tables 1.12 : hut1\_12v2.pdf

<https://usb.org/document-library/hid-usage-tables-112>

A Fast Compact CRC5 Checker for Microcontrollers : crc5check.pdf

<https://www.michael-joost.de/crc5check.pdf>

Cyclic Redundancy Checks in USB : crcdes.pdf

<https://usb.org/document-library/cyclic-redundancy-checks-usb>

<https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/>

<https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/hidsdi/>

You can download all these files from my website if you can not find them on the official website of Microchip and USB organization. Some files in the package have been translated into Chinese.

<http://wiki.geniekits.com/lib/exe/fetch.php?media=downloads:ya-vusb-refs.zip>

The USB Vendor ID I used belongs to my former employer, FEITIAN Technologies Co., Ltd., <https://www.ftsafe.com>, and I picked a random PID for this project. If you need a dedicated Vendor ID, you can check out the official website of USB organization, <https://usb.org/developers>, for more information about specific VID applying.

All source codes written by me are distributed under the MIT license. Any other references included in the package are published on the Internet and copyright by the original authors. The only purpose I put these references on my website for download is for convenience.

## Details of the source codes

```
/*-----*/
/*- <1> Digests of Reference docs -*/
/*-----I'm NOT gorgeous dividing line-----*/
```

- Power supply of USB: VBUS=4.75V - 5.25V
- High level output of D+/D-: VOH=2.8V - 3.6V
- High level input of D+/D-: VIH=2.0V (minimum)
- Low level output of D+/D-: VOL=0.0V - 0.3V
- Low level input of D+/D-: VIL=0.8V (maximum)

All above is listed on page 178, section 7.3.2 of USB2.0 Specification. Comparing with TABLE24-9 and TABLE24-10 on page 229 and page 232 of DS70265E, we know that we can connect USB D+/D- to GPIO pins of dsPIC33 chip directly and read or generate signals of USB.

- Single End 1 (SE1) of D+/D-: D+ and D- > 0.8V (RA1-RA0 = 11b)
- Single End 0 (SE0) of D+/D-: D+ and D- < 0.3V (RA1-RA0 = 00b)
- Differential 1 (DF1) of D+/D-: D+ > 2.8V and D- < 0.3V (RA1-RA0 = 01b)
- Differential 0 (DF0) of D+/D-: D+ < 0.3V and D- > 2.8V (RA1-RA0 = 10b)

All above is listed on page 144, section 7.1.7 of USB2.0 Specification. I use RA0 to connect D+ and RA1 to connect D-. The binary value in the parentheses is the signal read or output through GPIO A. For a low speed device, differential 0 (DF0) corresponds the J-state as well as the IDLE of D+/D-, and differential 1 (DF1) corresponds the K-state.

According to the section 7.1.5.1 on page 141 of USB2.0 Specification, we can see the detailed circuit of the connection between the HOST and USB device. The D+/D- of HOST or HUB port is pulled down by two 15K resistors. On the D- pin of a low speed device, there is a 1.5K pull-up resistor called Rpu, which is connected to 3.3V power. Device insertion and its speed could be detected by the HOST or HUB through this pull-up resistor.

When the device is plugged into the HOST port, the HOST or HUB will send a "BUS RESET" signal to the device. It is a "SE0" lasted for 10mS at least. It is important to distinguish the signal "BUS RESET" from "Keep-alive". Please refer to the section 7.1.7.5 on page 153 and section 11.8.4.1 on page 332 of USB2.0 Specification. Then the HOST will send a "SETUP" request to the "ENDPOINT 0" of the device through the initial device address (0000000b). A new device address will be assigned by the HOST after a couple of interactions between the HOST and DEVICE.

A completed data transmission is always split into three steps. First step, the HOST or HUB sends a "TOKEN" packet to the device. The direction is from HOST to DEVICE. Second step, transmits the "DATA" packet from HOST to DEVICE or reversely. Third step, the data receiver has to send a "HANDSHAKE" packet after the "DATA" packet. Each step is led by a "PID" byte. For the "TOKEN" packet, the PIDs include "SETUP/OUT/IN". In these three PIDs, "SETUP" and "OUT" indicates that the next "DATA" packet will be sent to the device. PID "IN" indicates that the device must send back a "DATA" packet. The PIDs for "DATA" packet include "DATA0" and "DATA1". The PIDs for "HANDSHAKE" packet include "ACK/NAK/STALL". The definition of PIDs is listed in Table8-1 on page 196 of USB2.0 Specification.

A "TOKEN" packet consists of three bytes, including one "PID" byte and two bytes of "device address/endpoint number/CRC5 code". The initial device address is zero and will be replaced by a new address assigned by the host. Endpoints are used to organize different transmission type. For now we only use the "ENDPOINT 0" to transmit data. Please read the section 8.3.2 on page 197 of USB2.0 Specification to get more information about device address and

endpoint and I will write more about it. The definitions of "TOKEN" packet is listed in section 8.4.1 on page 199 of USB2.0 Specification.

A "DATA" packet has up to 11 bytes because we use 8 bytes valid data length at most. In these 11 bytes, a PID byte and two bytes CRC16 verify code are combined with 8 bytes DATA. If the valid DATA length is shorter than 8 bytes, the CRC16 code must follow the DATA bytes in rapid sequence, making a shorter bytes stream which is called "short packet". The valid data length is allowed to be zero and the DATA packet just consists of a PID byte and two bytes of CRC16. This type of packet is called "empty packet" or "zero length packet". Please read the section 8.4.4 on page 206 of USB2.0 Specification to get more information. The two PIDs, DATA0 and DATA1, are used alternatively. If the previous DATA packet which is led by "DATA0" is received successfully, the next DATA packet will be led by "DATA1" and a new DATA packet will be led by "DATA0" again.

There is only one PID byte in a "HANDSHAKE" packet. In my code only "ACK" and "NAK" are processed. The device sends an "ACK" HANDSHAKE when it receives a "SETUP" or "OUT" TOKEN and the DATA packet attached to the TOKEN, that means the TOKEN and DATA were received properly and the next transmission is allowed to continue. If the device sends a "NAK" HANDSHAKE to the host, that means the TOKEN and DATA are not received successfully. The host MUST resend TOKEN and DATA when it gets a "NAK". Please refer the section 8.4.5 on page 206 of USB2.0 Specification. There is an exception we have to notice that if the host sends a "SETUP" TOKEN and the "DATA" packet to the device, the device MUST accept the TOKEN and DATA then responds an "ACK". It is NOT allowed to respond "NAK" or "STALL" to a "SETUP" TOKEN. Please refer the section 8.4.6.4 on page 209 of USB2.0 Specification. When the host sends an "IN" TOKEN to the device and the "DATA" packet of device is NOT ready, the device has to send "NAK" to the host and the "IN" TOKEN will be sent repeatedly. When the "DATA" packet of device is ready, the device sends this "DATA" then waits for an "ACK" or "NAK" response from the host. "ACK" response means the "DATA" packet has been received successfully and next transmission is allowed. "NAK" response requests the device to resend the "DATA" packet.

The "PID" byte is NOT counted in the byte-stream that needs to be verified with CRC5 or CRC16 algorithm. Low nibble of "PID" is defined in Table8-1 on page 196 of USB2.0 Specification and the high nibble is the "ones' complement" of low nibble. For example, the PID "SETUP" is defined as "1011b" and the "ones' complement" is "0100b", then the completed "PID" byte is "01001011b" (0x2D).

Every transmission of USB is generated by the host and responded by the device. Each step of transmission (TOKEN/DATA/HANDSHAKE) is led by a "SYNC" byte and ended by an "EOP" (End Of Packet). The value of "SYNC" byte is "10000000b" (0x80). The real bit-stream is "00000001b" because of "right justified". The coding format of USB bit-stream is "NRZI". That means if the bit is 0, the state of D+/D- will be switched from J-state to K-state or vice versa. If the bit is 1, the state of D+/D- MUST keep stable. Because the initial state of D+/D- is J-state, the bit-stream "00000001b" will be coded as "KJKJKJKK". Please refer the section 7.1.8 on page 157 and section 7.1.10 on page 159 of USB2.0 Specification. The "EOP" consists of two "SE0" plus a "J-state". Please refer the Table7-2 on page 145 and Table7-10 on page 183 of USB2.0 Specification.

If there are six continuous "1" in a "NRZI" bit-stream, a.k.a "KKKKKKK" or "JJJJJJ", then a bit "0" MUST be inserted into the bit-stream. That means the state of D+/D- MUST be switched. This rule is called "BIT-STUFF". This zero bit which is inserted into the bit-stream is NOT a valid data bit and MUST be omitted by the receiver. Please refer the section 7.1.9 on page 157 of USB2.0 Specification.

All data transmissions are organized into frames at 1mS interval. In my option, a completed transmission is NOT allowed to span the boundary between frames. The "TOKEN" and "DATA" are never split into two frames. If the remaining time in previous frame isn't enough for a completed transmission, it will be ignored and transmission will start in next frame. Please refer to the section 7.1.12 on page 159 and section 5.3.3 on page 36 of USB2.0 Specification. We MUST notice that



the "keep-alive" signal is always generated at the beginning of each frame. This signal consists of 2bit "SE0" and 1bit "J-state". It's same as "EOP" signal. Please refer to the section 11.8.4.1 on page 144 and section 11.8.4.1 on page 332 of USB2.0 Specification.

To the PIC24F/dsPIC33 chip, most of instructions executes within one machine cycle. Unconditional branch instructions needs two machine cycles. Conditional branch instructions executes within two machine cycles when the branch condition is fulfilled and one cycle when the condition is not fulfilled. In addition, the built-in interrupt controller has five cycles latency and the instruction "RETFIE" needs three cycles. Please refer to the section 3.2.1 on page 39 and section 4.3 on page 62 of DS70157F and the description of "Interrupt Controller" on page 3 of DS70265E.

The unit of time we use in this document is not only "S/mS/uS", but also "bit". For a low speed USB device, 1bit transaction needs 1/1.5 microsecond (0.667 uS). That corresponds ten machine cycles of the CPU running at 15 MIPS.

```
/*-----*/
/*- <2> Pull-up Resistor on D- -*/
/*-----I'm NOT gorgeous dividing line-*/
```

In the schematic diagram we can see the pull-up resistor on D- is controlled by RB4 pin. The device insertion will be detected by the host or hub when the firmware generates a high level through RB4. That means we can do some complicated initialization in the firmware before it enable the pull-up resistor. In my source code file "sie.s", I output a high level through RB4 in function "\_\_user\_init()". It can be moved into function "setup()" in source file "main.c". At any time the firmware generates a low level or high-z through RB4, the host or hub will detected a "device removal" event. We don't have to unplug the device physically. Please refer to the section 7.1.7.3 on page 149 of USB2.0 Specification.

If we connect the pull-up resistor to 3.3V directly, the source file "sie.s" doesn't need to be modified unless the RB4 pin needs to be used for other purposes. I haven't tried to use "RAx" pin to control the pull-up resistor. Please check the source code to confirm that all "RAx" pins except "RA0/RA1" are not modified accidentally by the firmware.

The first chip with USB function which I used is CY7C63001 produced by CYPRESS Co. This chip uses 5V power supply and the pull-up resistor on D- pin is 7.5K connected to 5V directly. This 7.5K resistor and 15K pull-down resistor of the host port bias D- pin at 3.3V. It meets the standard of USB specification.

In addition, in the function "\_\_user\_init()" I disabled analog input and open drain output on all pins. It is not necessary so you can enable them if you need to.

```
/*-----*/
/*- <3> Device Insertion -*/
/*-----I'm NOT gorgeous dividing line-*/
```

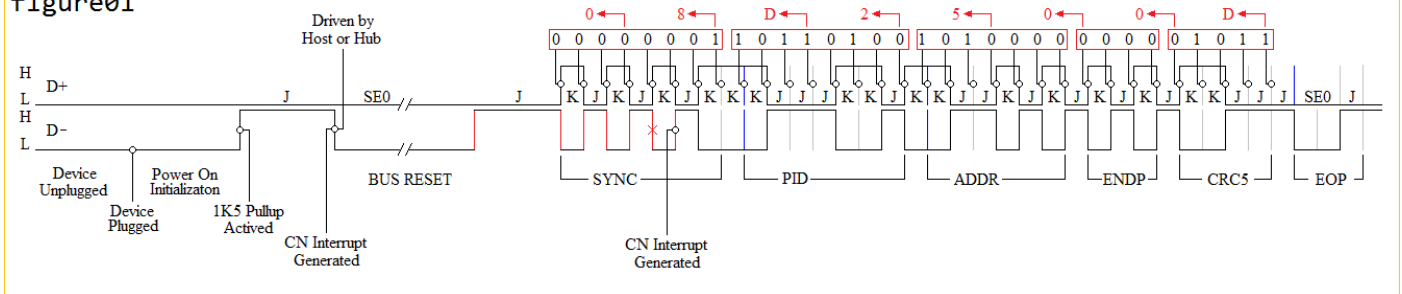
When the device is inserted into the USB port of host or hub and detected, the host or hub will send a "BUS RESET" signal to the device. Because the level of D- pin is pulled up to high by the 1.5K pull-up resistor when the device is in IDLE state and driven to low during the "BUS RESET" period, I enable the "Change Notification (CN3)" interrupt at D- (RA1) pin. Please refer to the section 10.4 on page 10-7 of DS70193C. If you need to enable other interrupts, you MUST set the priority of CN interrupt to the highest.

It MUST be noticed that when a low speed device is detected, the host or hub will send "Keep-alive" signal automatically to the device at the beginning of each data frame. The firmware MUST distinguish the "BUS RESET" from "Keep-alive" in the CN interrupt service routine. If you enable CN2 interrupt at D+ (RA0) pin, the "BUS RESET" and "Keep-alive" will be ignored automatically. The code complexity and interrupt frequency will be reduced.

The bit-stream generated by the host or hub after device insertion is demonstrated in figure01. After the "BUS RESET"

signal, the first byte sent by the host or hub is "SYNC". Each level switching of D+/D- (the edges rendered with red color) will trigger the CN interrupt. The firmware tries to capture the last three bit (JKK) of "SYNC" in the CN interrupt service routine. When the bit "JKK" is captured successfully, the interrupt service routine will be continued to capture the whole bit stream until the "EOP".

figure01



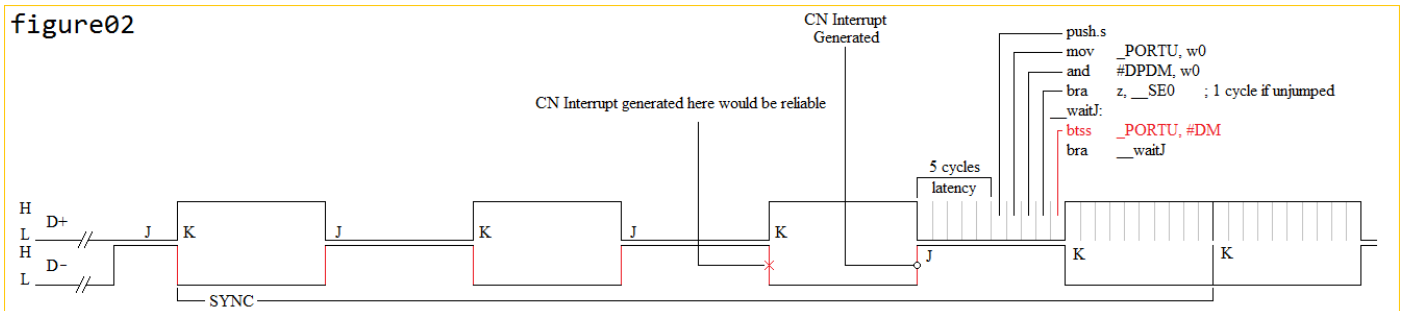
```

/*-----*/
/*- <4>  Capture the SYNC Byte                                -*/
/*-----I'm NOT gorgeous dividing line-----*/

```

When the CN interrupt service routine is entered, the register W0-W3 will be pushed into the shadows register by the first instruction "push.s" then a "mov \_PORTU, w0" instruction is executed to read the level of D+/D- pin. "\_PORTU" has been defined as "PORTA". Because the interrupt controller has 5 machine cycles latency, the first instruction "push.s" is executed at the 6<sup>th</sup> cycle and the level of D+/D- is read at the 7<sup>th</sup> cycle. One bit corresponds 10 machine cycles of CPU. That means it's a little bit late to read the level of D+/D- at the 7<sup>th</sup> cycle but it's still reliable. The purpose of reading the level of D+ and D- simultaneously is to identify that whether the state of D+/D- is a "SE0" or not. If it is a "SE0", the firmware will branch to the label "\_\_SE0:" and distinguish the current "SE0" is a "BUS RESET" signal or a "Keep-alive".

figure02



In order to capture the last three bits "JKK", the firmware uses a loop to make it sure that the current state is a "J-state". In figure02 we can see that if the CN interrupt is triggered at the rising edge of the "J-state" (marked by a black o), then the instruction "btss \_PORTU, #DM" will be executed at the last cycle of this bit. That means it is difficult to capture this "J-state" reliably. It is reasonable to consider that the CN interrupt should be triggered at the rising edge of previous "K-state" (marked by a red x), then waiting for the "J-state" in a loop labeled with "\_\_waitJ".

The codes for capturing the next "K-state" are little bit odd. It is helpful to get a determinate latency. In figure03 and figure04 we can see the relation between the executing time of "btsc \_PORTU, #DP" and the latency. The latency is three or four machine cycles.

figure03

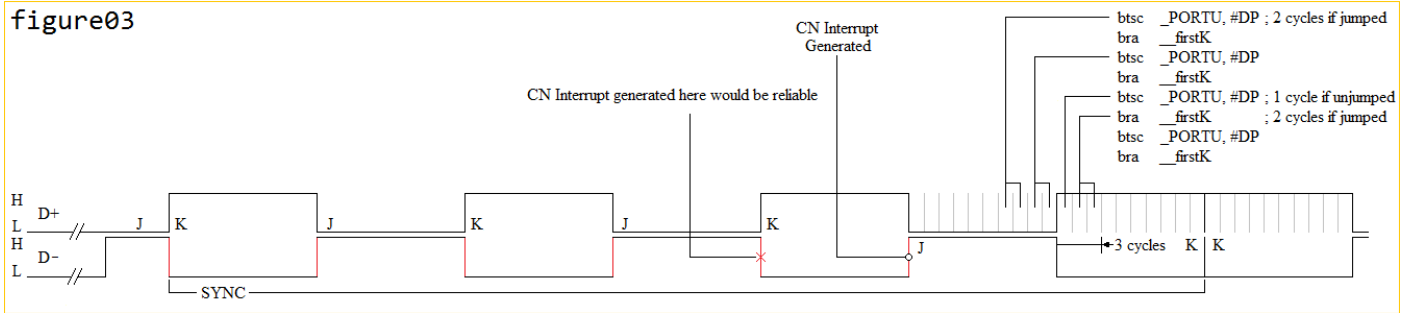
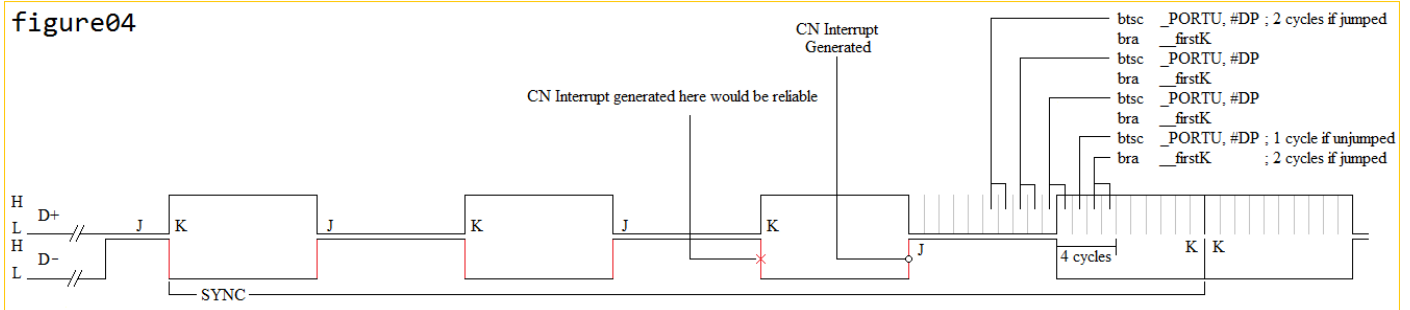


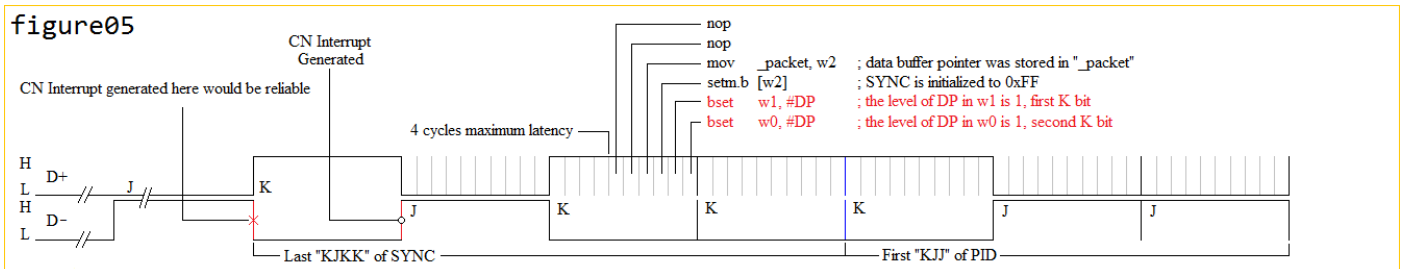
Figure03 shows that the instruction "btsc \_PORTU, #DP" is executed at the 1<sup>st</sup> cycle of the first K-state. It generates three cycles latency. Figure04 shows that the instruction "btsc \_PORTU, #DP" is executed at the 2<sup>nd</sup> cycle of the K-state and four cycles latency is generated.

figure04



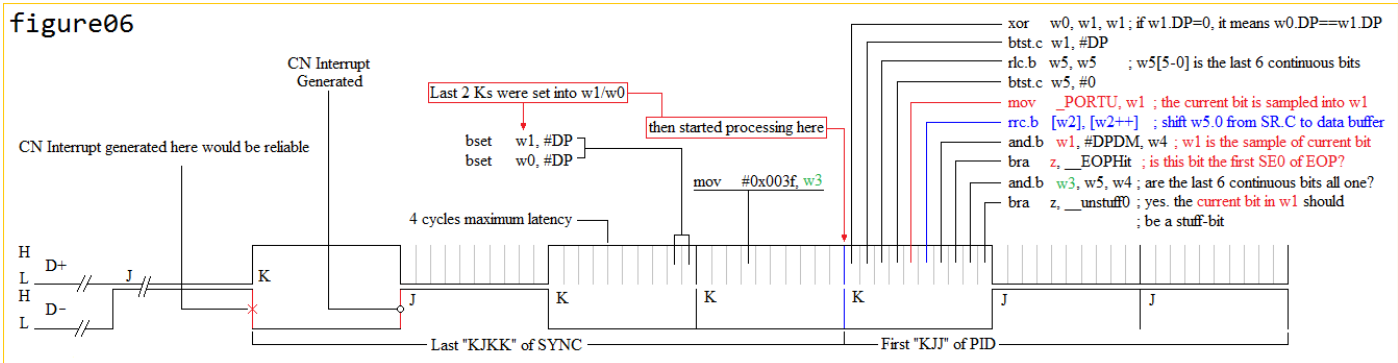
When the last three bits "JKK" is captured successfully, the firmware will receive a completed bit stream including the "SYNC" byte. Because the "SYNC" is a constant byte and the first five bits "KJKJK" has been missed, it should be written into memory buffer directly. What I did is a little bit different. The first byte in the memory buffer is initialized to "0xFF" then I wrote a piece of code with ten instructions to calculate the last bit of "SYNC", which equals "1" represented by the last two samples "KK" of "SYNC". The real value of last bit of "SYNC" is "0" and right shifted into the first byte of buffer. That means the "SYNC" byte in memory buffer is "0x7F". It is "ones' complement" code of "0x80". All bytes transmitted on the USB bus will be gathered into memory buffer as "ones' complement" code. The diagram figure05 shows the preparation work before calculating the last bit of "SYNC" -- Writing a "K-state" into register w0 and w1 directly.

figure05



The figure06 shows the ten instructions which are used to calculate the last bit. A "XOR" operation is used to determine whether the two samples of D+/D- are identical or not. If they are identical, the result of "XOR" will be zero. Because it is opposite to the rules of "NRZI" coding, all bytes we received are "ones' complement" code. This bit is sent to bit C of SR register then shifted into register w5 and memory buffer pointed by [w2]. Because I want to sample the current bit at the 5<sup>th</sup> cycle (the central point of each bit), the instruction "mov \_PORTU, w1" is inserted between "btst.c w5, #0" and "rrc.b [w2], [w2++]". This piece of code calculates the last bit of current byte and sample the first bit of next byte. That's why the second operand of instruction "rrc.b" is "[w2++]".

figure06



/\*-----\*/  
/\*- <5> bit stuff processing -\*/  
/\*-----I'm NOT gorgeous dividing line-----\*/

If a "stuff-bit" is detected at the ninth or tenth cycle (the low six bits of register w5 is 000000b), then the firmware will branch to the label "\_\_unstuff0:" and continue. The sample of the previous bit in register w0 will be discarded. The sample of the current bit in register w1 will be moved into w0 and the firmware samples the next bit into register w1. We know the first K-state of PID byte which is showed in figure07 is not a "stuff-bit". However, this piece of code is used to receive the whole bit stream so it will branch to "\_\_unstuff0:" when a "stuff-bit" in the next bytes is captured.

figure07

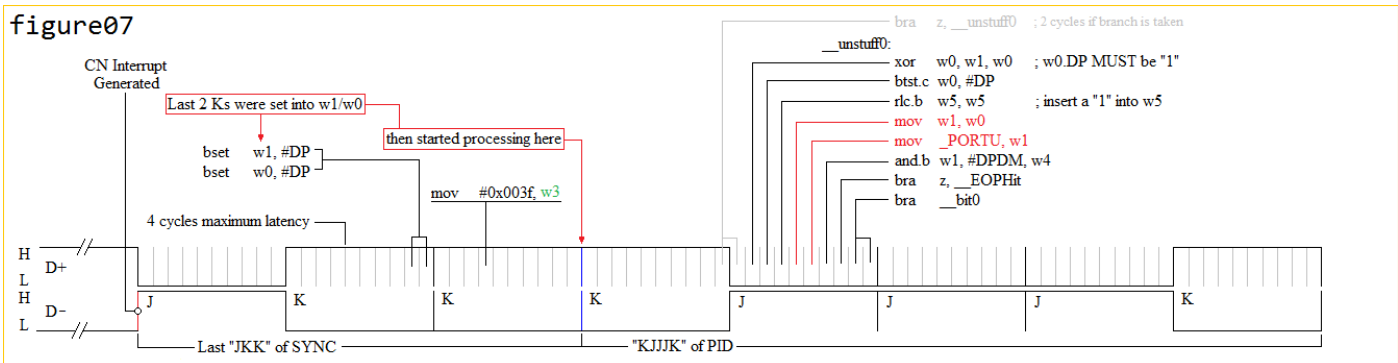


Figure07 shows the details of bit-unstuff. At the label "\_\_unstuff0:", the first three instructions inserts a "1" into the least bit of register w5. It could be simplified with two instructions -- a "bset w5, #0" and a "nop" -- then the instruction "mov \_PORTU, w1" could be executed at the 5<sup>th</sup> cycle (It is executed at the 6<sup>th</sup> cycle now). The last instruction "bra \_\_bit0" needs two cycles to be executed.

In figure07 we can see that the first K-state of PID is assumed to be a stuff bit and copied into register w0 when the instruction "mov w1, w0" is executed. The current J-state is sampled into register w1. The bit marked by a BLUE vertical bar in figure07 has not been calculated.

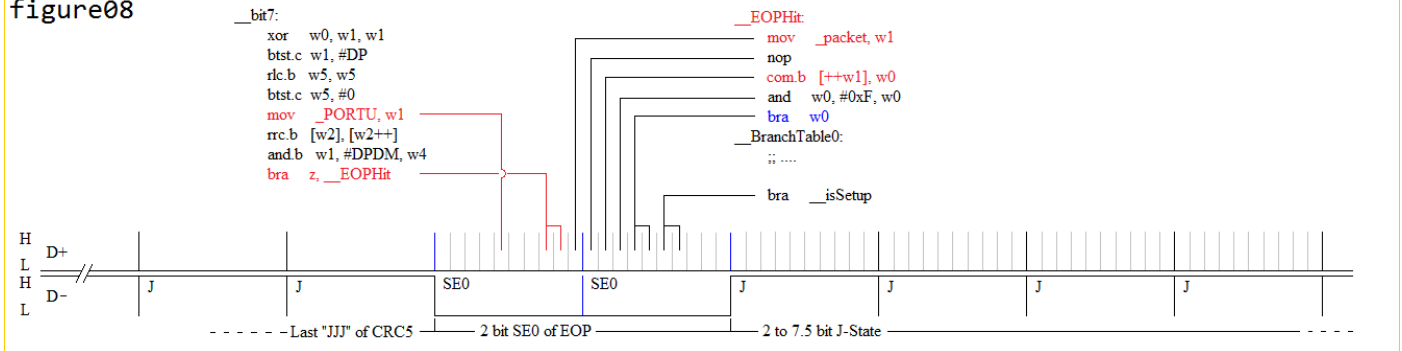
/\*-----\*/  
/\*- <6> The Last bit of a Byte and the EOP -\*/  
/\*-----I'm NOT gorgeous dividing line-----\*/

The firmware tests the sample if it is a "SE0" or not after sampling the bit0 AND bit1 because the "USB HUB" has a special feature which is called "EOP dribble". Please refer to the section 7.1.9.1 on page 157 of USB2.0 Specification. This is a simple way to identify the ending of the bit stream but seems to be not reliable. The data transmission fails occasionally when the device is connected to a HUB. Please refer to the section "Known BUGs" at the end of this document. The procedure is a little bit different when the bit7 (last bit of a byte) is sampled. The instruction "bra z, \_\_unstuff7" is executed at the 8<sup>th</sup> but not the 10<sup>th</sup> cycle because I MUST put the instruction "bra \_\_bit7" at the 9<sup>th</sup> and 10<sup>th</sup> cycle. There MUST be a

LOOP to receive the whole bit stream. That makes the codes at the label "`__unstuff7:`" different from others. I don't like to use the "DO loop" feature of dsPIC33 because it is impossible to be ported to PIC24F series chip. I don't make a figure to demonstrate this piece of code. You can read the nine instructions below the label "`_bit6:`" in the source file "`sie.s`".

The firmware uses a "Branch Table" to process each PID when the first "SE0" of "EOP" is captured. The "Branch Table" is helpful to get a consistent latency. The figure08 shows that piece of code with "`__BranchTable0:`".

figure08



The instruction "`mov _packet, w1`" fetches the head address of the current memory buffer into register `w1`. Because all bytes in the buffer are "ones' complement", the instruction "`com.b [++w1], w0`" fetches and negates all bits of the PID byte.

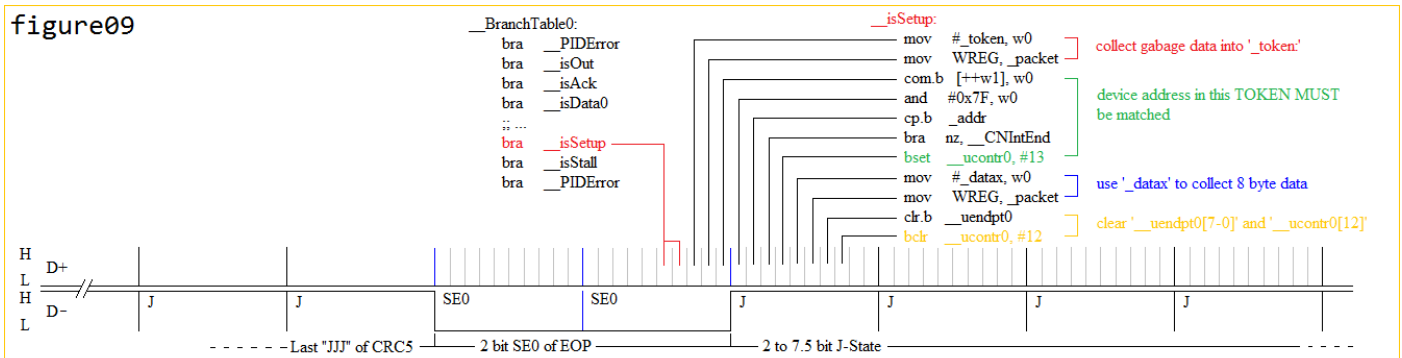
```

/*-----*/
/*- <7>  Token SETUP and OUT Processing                                     -*/
/*-----I'm NOT gorgeous dividing line-----*/

```

There are three steps to process token "SETUP". First, compare the device address in token with the address maintained by the firmware. If the device address in token is matched with the address stored in variable "`__addr:`", the firmware will set the bit13 of "`__ucontr0:`". That means the data attached to this "SETUP" is allowed to be received. If the device address is not matched, the firmware will exit CN interrupt service routine directly and ensure that all data packet received after interrupt exiting MUST be stored into the buffer "`__token:`". Because USB HUB will broadcast all transmissions to all downstream ports, the firmware MUST ensure that all data packets without correct device address are stored into the buffer "`__token:`" and won't destroy the buffer "`__datax:`" and "`__datay:`". Second, the buffer "`__datax:`" is reassigned to the variable "`__packet:`" (the current buffer pointer). All eight bytes data packet attached on the token "SETUP" MUST be sent with a PID "DATA0" and stored into "`__datax:`". Third, initialize the low byte of variable "`__uendpt0:`" with "0x00". That means a valid "Control Transmission" is starting now. At the same time, the bit12 of variable "`__ucontr0:`" is cleared to indicate that the data attachment of the next token "IN" or "OUT" will be transmitted with PID "DATA1". The figure09 shows how the firmware handles a token "SETUP".

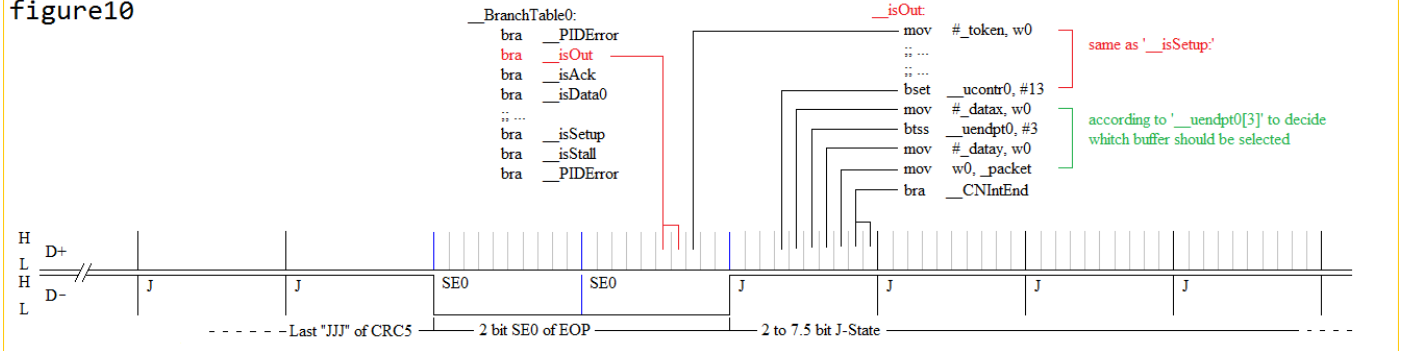
figure09



There are two steps to process a token "OUT". First, compare the device address just like processing the token "SETUP". Second, according to the bit3 of variable "`__uendpt0`" the firmware determines which memory buffer --

"\_\_datax:" or "\_\_datay:" -- should be used to store the data attached on this "OUT". More specifically, the buffer "\_\_datax:" has been occupied if the "\_\_uendpt0[3]" equals "0" and the firmware MUST use the buffer "\_\_datay:". When the "\_\_uendpt0[3]" equals "1", the buffer "\_\_datay:" has been occupied and the firmware MUST use the buffer "\_\_datax:". Here the firmware doesn't modify the "\_\_uendpt0[7-0]". These bits are modified only when the data attachment is received correctly and an "ACK" is sent to the host.

figure10

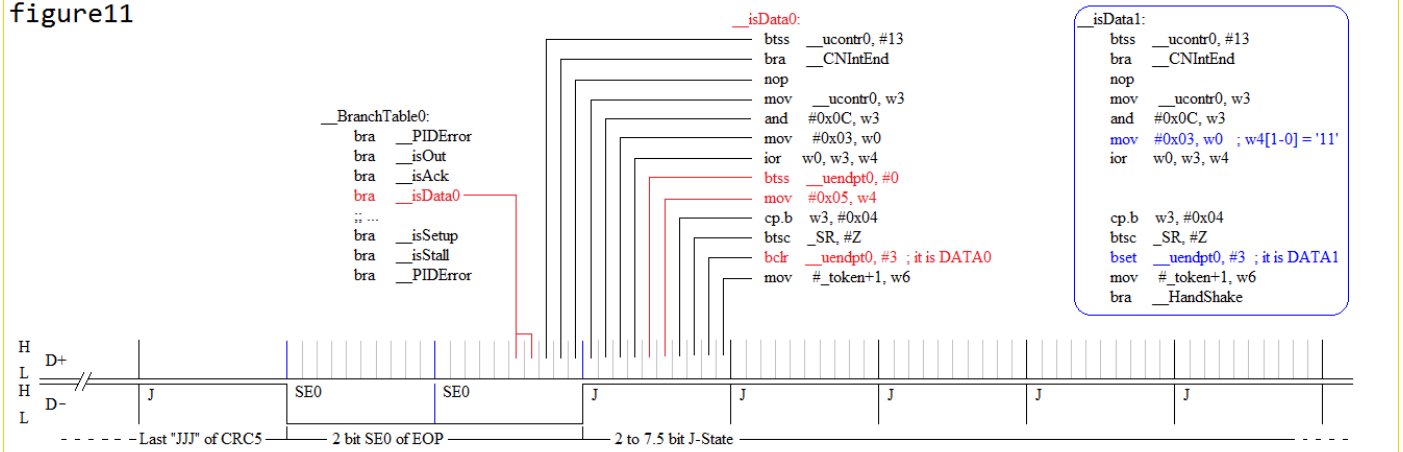


The CN interrupt service routine will be exited directly when the token "SETUP" or "OUT" is processed. The firmware doesn't receive the data attachment in the same interrupt context. It is NOT a good design and wastes the memory. The size of buffer "\_\_token:" has to be increased to 12 bytes in order to accommodate the longest transmission (SYNC + PID\_8bytes + CRC16) broadcasted by the HUB. A bit variable "\_\_ucontr0[13]" has to be used to associate the token and its data attachment. (Note: The "\_\_ucontr0[13]" will be set if the device address is matched.) Another way to simplify the code is waiting and capturing the data attachment directly when a token is processed. We can use the three bit of EOP and the first 6bit (KJKJKJ) of the next "SYNC" and a group of J-state (6.5bit maximum) between "EOP" and the next "SYNC" to try to do the "CRC5" verification then branch back to the label "\_\_waitK:" to continue to receive the data attachment. Please refer to the section 7.1.18.1 on page 168 of USB2.0 Specification to get more information about the time interval between an "EOP" and the next "SYNC" byte.

```
/*-----*/
/*- <8>  PID DATA0 and DATA1 Processing                                     -*/
/*-----I'm NOT gorgeous dividing line-----*/
```

The processing to a PID "DATA0" is different from PID "DATA1". A "DATA1" is only attached to a token "OUT" but a "DATA0" might be attached to a token "OUT" or "SETUP". That means the firmware MUST distinguish these two different types of transmission in order to give a correct report about the current token when the data packet is received. This is another inconvenience splitting the "data" stage from the "token" stage.

figure11





In the figure11 we can see the difference between a "DATA0" and "DATA1". w4[1-0] will be set to "01" when a token "SETUP" is processed successfully. w4[3-2] is "01" as well. That means an "ACK" has been sent to the host. If a token "OUT" is received successfully, w4[1-0] will be set to "11" and w4[3-2] will be set to "ACK" or "NAK" according to the indicator in the variable "\_\_ucontr0[3-2]".

It depends on the variable "\_\_uendpt0[0]" to determine what type of token – "SETUP" or "OUT" – takes the current DATA0 packet. The variable "\_\_uendpt0[7-0]" is always cleared when a token "SETUP" is received. That's why we use an instruction "btss \_\_uendpt0, #0" to do this judgment.

```
/*-----*/
/*- <9> Details of variables __uendpt0 and __ucontr0 -*/
/*-----I'm NOT gorgeous dividing line-----*/
```

There are two important variables defined in the module "sie.s" – "\_\_uendpt0" and "\_\_ucontr0". The variable "\_\_uendpt0[1-0]" is used as a flag that indicates the type of the current token. When the host or hub sends a token "SETUP" and its data attachment, the firmware will respond an "ACK" and set "\_\_uendpt0[1-0]" to "01". The "\_\_uendpt0[2]" is simultaneously set to "1" to indicate that an "ACK" has been sent. Because the PID of data attached to a token "SETUP" is always "DATA0", the "\_\_uendpt0[3]" will be set to "0". "\_\_uendpt0[7-4]" indicates the valid byte length of the data attachment. It is usually set to "8".

When the host or hub sends a token "OUT" and its data attachment, the firmware will determine which buffer (\_\_datax or \_\_datay) should be used for the data attachment according to "\_\_uendpt0[3]". The variable "\_\_uendpt0" won't be modified at this time. When the data attachment is received, the firmware will send a handshake according to the value of "\_\_ucontr0[3-2]" and "\_\_ucontr0[3-2]" is initialized to "10" at power on and "BUS RESET" period. The initial value "10" means sending a "NAK" to the host. The value of "\_\_uendpt0[7-0]" won't be modified when the firmware sends "NAK" repeatedly.

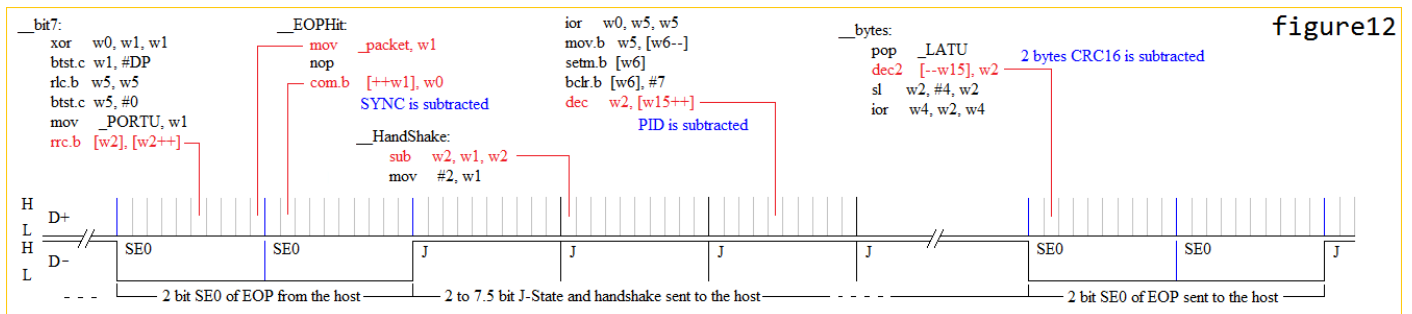
If we are ready to handle the data attached to the token "OUT", we can set the "\_\_ucontr0[3-2]" to "01". When the host resend the token "OUT" and data attachment, the firmware will send an "ACK" to the host and set "\_\_uendpt0[2-0]" to "111". That means the token "OUT" and its data attachment has been received and an "ACK" has been sent successfully. The "\_\_uendpt0[3]" is used as a flag which indicates that the PID of data attached to the current token "OUT" is "DATA0" or "DATA1". If it is a "DATA0", the "\_\_uendpt0[3]" equals "0". The "\_\_uendpt0[7-4]" indicates the byte length of data attachment and is between 0 and 8. The "\_\_ucontr0[3-2]" will be reset to "10" when an "ACK" is sent successfully. The firmware will respond "NAK" until we set the "\_\_ucontr0[3-2]" to "01" again.

When the host or hub sends a token "IN", the firmware will send a handshake specified by the variable "\_\_ucontr0[1-0]". It is initialized to "10" at power on and "BUS RESET" period. That means sending a "NAK" to the host. The "\_\_uendpt0" won't be modified when a "NAK" is sent repeatedly. If the data sent to the host is ready, we can set the "\_\_ucontr0[1-0]" to "01" then the data packet will be sent when the host sends a token "IN" again. The firmware will determine which PID ("DATA0" or "DATA1") should be used according to "\_\_ucontr0[12]". When "\_\_ucontr0[12]" equals "0", the "DATA1" will be used as the PID of current data packet. It is contrary to the "\_\_uendpt0[3]". We must notice that the firmware always sets the "\_\_ucontr0[12]" to "0" when a token SETUP is received because the PID of the first data packet attached to the token "OUT" or "IN" is always "DATA1". The host will send an "ACK" or "NAK" to the device when the data packet is received. The firmware will toggle the "\_\_ucontr0[12]" and reset "\_\_ucontr0[1-0]" to "10" when an "ACK" is received. If a "NAK" is received, the firmware will set the "\_\_ucontr0[1-0]" to "01" and resend the buffered data packet automatically.

The "Control Write" (SETUP+OUT+OUT+.....+OUT+IN) transmission procedure depends on the token type represented by "\_\_uendpt0[1-0]". When a token "SETUP" is received, the firmware sets "\_\_uendpt0[1-0]" to "00". The

firmware could determine the token type through the bit value of "`__uendpt0[0]`" when the data attachment is received. If the "`__uendpt0[0]`" equals "0", the current token is a "SETUP". The firmware sends an "ACK" and set "`__uendpt0[1-0]`" to "01". When the first token "OUT" and its data attachment (marked by "DATA1") comes, the firmware will send an "ACK" and set the "`__uendpt0[1-0]`" to "11". In another word, the "`__uendpt0[1-0]`" MUST be "00" or "11" when an PID "DATA0" is detected. That's why the firmware can determine the token type by only the least bit of "`__uendpt0[1-0]`".

It is complex to maintain the value of "`__uendpt0[10]`" and "`__uendpt0[7-0]`". When a token "SETUP" is received, the "`__uendpt0[7-0]`" will be cleared. Please refer to the codes blow the label "`__isSetup:`". If a data attachment marked by a "DATA0" or "DATA1" is received, the firmware will clear or set the "`__uendpt0[3]`" if an "ACK" is responded. Please refer to the codes blow the label "`__isData1:`" and "`__isData0:`". When a HANDSHAKE starts to be sent, the firmware will clear "`__uendpt0[7-4,2-0]`" and keep "`__uendpt0[3]`" unchanged if the HANDSHAKE is an "ACK". Please refer to the codes blow the label "`__Sending:`". After sending the HANDSHAKE, the firmware will set the "`__uendpt0[10,7-4,2-0]`" according to the transmission status. Please refer to the codes blow the label "`__bytes:`".



The figure12 shows that how the firmware calculates the valid byte length of the data attachment sent by the host. The codes is split into six pieces. When the data is received, the register w2 points to the tail of the whole data attachment including SYNC/PID/CRC16. The memory buffer head is fetched into register w1 (instruction: `mov _packet, w1`). Then register w1 plus 1 (instruction: `com.b [++w1], w0`) that means the SYNC byte is discarded. The result of "w2-w1" (instruction: `sub w2, w1, w2`) includes the PID and CRC16 only. The firmware subtracts the PID byte and pushes the result into stack (instruction: `dec w2, [w15++]`). This result will be popped into register w2 when a HANDSHAKE is sent and it is subtracted 2 bytes by the instruction "`dec2 [--w15], w2`" that means the CRC16 is discarded. Now the value of register w2 is the valid byte length of current data attachment.

```
/*-----*/
/*- <10>  Data Sending and bit stuff                                -*/
/*-----I'm NOT gorgeous dividing line-*/
```

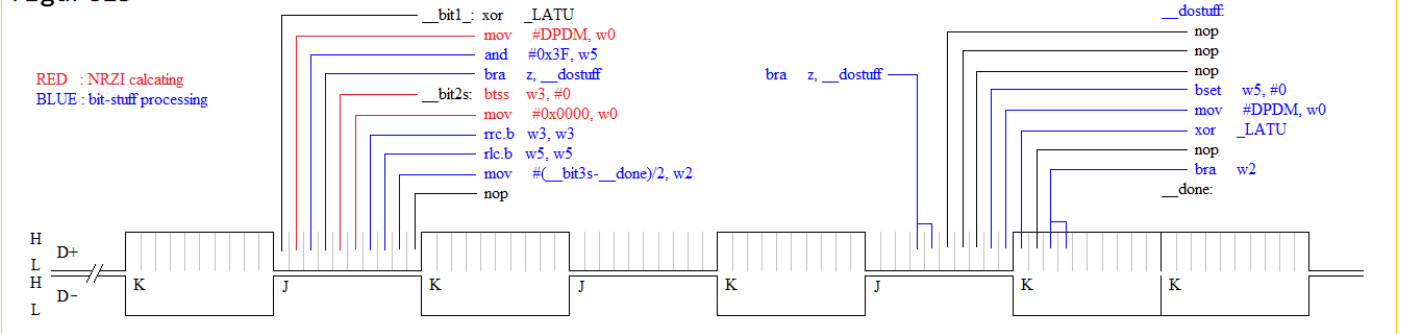
When the firmware sends data attached to a token "IN", the valid byte length (0 to 8 bytes) MUST be set into "`__ucontr0[7-4]`" first. The "`__ucontr0[12]`" is used to indicate the PID ("DATA0" or "DATA1") of current data attachment. This bit is cleared when a token "SETUP" is received and toggled when a HANDSHAKE "ACK" is received. The "`__ucontr0[15-13]`" is for internal use. The "`__ucontr0[11-8]`" is not used now.

It is very similar as the procedure of "bit-unstuff" to do "bit-stuff" when the firmware sends a bit stream to the host. The register w5 is used to track the "bit-stuff condition" (six continuous "1") within the bit stream. If the condition is detected, the firmware branches to the label "`__dostuff:`" and continue. A zero is inserted into the bit0 of w5 then the level of D+/D- is flipped. The instruction "`bra w2`" is used to return to the major routine. That is for reducing the code size and we MUST write a correct address into register w2.

In the figure13 we can see that an instruction "`xor _LATU`" is used to generate the signals of D+/D- through the "`_PORTU`". Flipping or remaining the level of D+/D- is controlled by the value of the register w0.

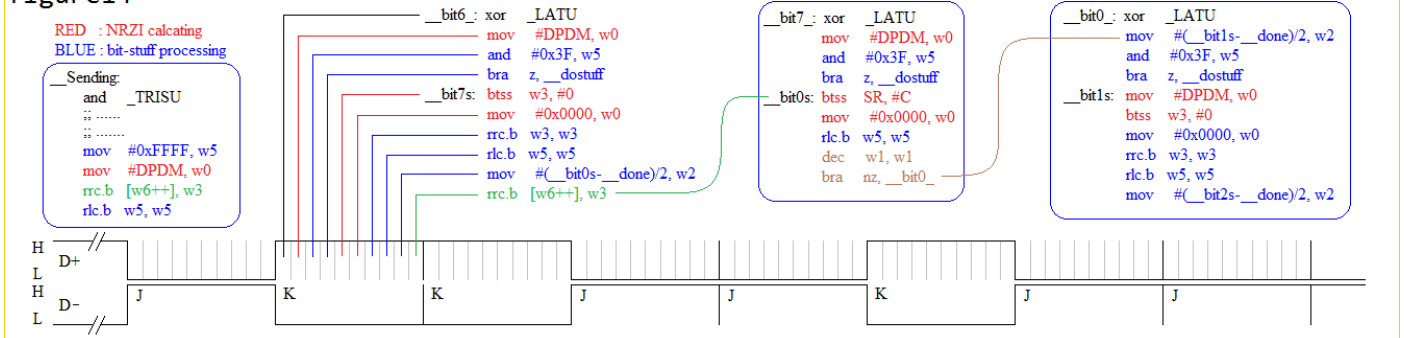


figure13



It is a little bit different from the sending procedure of bit1 to bit5. When the bit6 is sent, the firmware uses the last cycle to fetch a new byte from memory buffer into register w3. Because the firmware always uses instruction "rrc.b [w6++], w3" to fetch the data, there are only seven bits (bit1-bit7) fetched into register w3 and bit0 is shifted into register SR[C]. That means the firmware has to prepare the NRZI code for the bit0 of next byte according to the value of SR[C] which MUST NOT be destroyed by all instructions below "rrc.b [w6++], w3" including the codes of procedure "\_\_dostuff:". The firmware doesn't prepare the return address in register w2 for the "\_\_dostuff:" procedure of the bit0 of next byte because three cycles are needed to form a LOOP. The return address is made up when the bit0 of next byte is sent.

figure14



```

/*-----*/
/*- <11>  Token IN and HANDSHAKE from the Host -*/
/*-----I'm NOT gorgeous dividing line-----*/

```

The procedure of token "IN" is simpler than "SETUP" and "OUT". We need to notice that there is an instruction "dec w1, w2" prior to the instruction "bra \_\_SendBytes". In this instruction the register w1 is the total byte length of the data sent to the host including "SYNC/PID/CRC16" and w2 is reduced by 1. That leads to an unexpected result -- the variable "\_\_uendpt0[7-4]" will be the valid byte length of the data sent to the host after all data are sent. You can replace this instruction with "mov #0x0003, w2" then the "\_\_uendpt0[7-4]" will be zero at the end of data sending.

The procedure of HANDSHAKE (ACK/NAK/STALL) processing might make you to be confused. The CN interrupt service routine exits when all data are sent to the host so a new interrupt will be triggered by the handshake of the host. Because the HUB always BROADCAST all data packets to all devices connected at the downstream ports, the handshakes sent to other devices will be received as well. Unfortunately there is only a PID byte and no device address in the handshake packet. I have to use the variable "\_\_uendpt0[2-0]" as a flag to determine which handshake is sent to my device. If the "\_\_uendpt0[2-0]" equals "110", the handshake should be sent to me because there was a data packet just sent to the host.

```

/*-----*/
/*- <12>  API functions in module sie.s -*/
/*-----I'm NOT gorgeous dividing line-----*/

```

If you want to coding base on the module "sie.s" but with C language, you can use the variable "\_\_uendpt0" and "\_\_ucontr0" directly. However, I highly recommend use the API functions built in the module "sie.s". There are seven API functions total and their C prototypes are listed below with a minor manual.

- `BYTE _usbGetSetup(BYTE * setup); /* Fetch the data attached to the token SETUP */`

The parameter "setup" is a pointer which will be written into register w0 when the function is called. The length of the buffer pointed by "setup" MUST be equal or greater than eight bytes. The parameter "setup" is allowed to point to an odd address. When it equals NULL, this API function will return a zero directly. This API function will copy eight bytes into the memory buffer pointed by "setup" when a token "SETUP" and its data attachment is received in advance or just return zero if the current token isn't "SETUP". You should poll this API function in the main task loop of your code.

- `void _usbLoadData(BYTE * _data, BYTE length); /* Upload the data to be sent to the host */`

The pointer parameter "\_data" MUST point to a buffer in which the data packet stored. This parameter will be written into register w0 by the C compiler. It is allowed to point to an odd address or to be NULL. The byte parameter "length" indicates the valid data length in bytes and will be written into register w1 by the C compiler. It MUST be equal or less than eight. When "\_data" equals NULL and "length" equals to zero, an "empty packet" will be sent to the host. The PID (DATA0 or DAT1) of the packet is determined according to the previous packet. This API function doesn't return until an "ACK" is received from the host and it doesn't have a strategy to deal with "time out".

- `BYTE _usbReadData(BYTE * _data, BYTE length); /* Fetch the data attached to the token OUT */`

The pointer parameter "\_data" MUST point to a buffer which is used to store the data. It will be written into register w0 by the C compiler. It is allowed to point to an odd address or to be NULL. When it equals NULL, the byte parameter "length" MUST be zero or the API function will return "-1" directly. The byte parameter "length" indicates the byte length of the buffer and it will be written into register w1 by the C compiler. When the "length" equals zero, this API function will wait an "empty packet" sent by the host led with whatever "DATA0" or "DATA1". When the "\_data" is not NULL, the parameter "length" will be compared with the real data length sent by the host and fill the buffer with minimum bytes. The unfilled buffer will keep the initial value.

- `void _usbSendZLP(void); /* Send an empty packet with PID "DATA1" to the host */`

In the "Status" stage of a "Control-write" sequence, you can invoke this API function to complete the current transmission. If you want to send an empty packet which isn't led by a particular PID "DATA1", you MUST invoke the function "\_usbLoadData" with parameter "(NULL, 0)".

- `void _usbWaitZLP(void); /* Waiting an empty packet sent by the host */`

This function equals "\_usbReadData(NULL,0)" and could be used in the final "Status" stage of a "Control-read" sequence. It doesn't matter that the PID of this empty packet is "DATA0" or "DATA1".

- `void _usbSetAddress(BYTE a) /* Invoke me when you receive the new device address assigned by the host */`

The byte parameter "a" MUST be the new address from the host. It will be written into register w0 by the C compiler. There are two things to be done in this function. First, the function "\_usbSendZLP" will be invoked automatically to complete the current "Control-write" sequence. Second, the variable "\_addr" is replaced with the new value of parameter "a".

- `void _usbSetConfig(BYTE c); /* Invoke me when you receive the new configuration assigned by the host */`


The byte parameter "c" MUST be the new configuration and it will be written into the register w0 by the C compiler.

The variable "\_conf" will be replaced with the new value of parameter "c" and the function "\_usbSendZLP()" will be invoked automatically to complete the current "Control-write" sequence.

```
/*-----*/
/*- <13> CRC5 and CRC16 Algorithm -*/
/*-----I'm NOT gorgeous dividing line-----*/
```


In this version of the firmware I didn't verify the CRC5 code bound on each TOKEN. I found a rapid CRC5 algorithm on the Internet -- <https://www.michael-joost.de/crc5check.pdf> -- and you may try to convert this piece of code from AVR assembly into dsPIC33 assembly. There is another regular algorithm -- [http://janaxelson.com/files/usb\\_crc.c](http://janaxelson.com/files/usb_crc.c).

The practical algorithm of CRC16 in my project comes from a thread which was posted on Microchip's forum -- <https://www.microchip.com/forums/m603309.aspx>. This CRC16 algorithm is used for not only USB but also other communication protocols. For example, the "Modbus" which is mentioned in this thread. There is another rapid CRC16 algorithm published at [https://www.modbustools.com/modbus\\_crc16.html](https://www.modbustools.com/modbus_crc16.html) but it hasn't been tested in my project.

**CRC-16 (Modubs) 0x8005 for PIC32**


Author
Essentials Only
Full Version
Post

**flamewatcher**
Saturday, September 24, 2011 6:21 AM (permalink)



**CRC-16 (Modubs) 0x8005 for PIC32**

★★★★★
0

I am having of trouble implementing the standard ANSI CRC-16 using the Pic32 DMA crc controller.  
The following code works in software:

```

UInt16 ComputeCRC16(const UInt8 * buf, int len)
{
    int j;
    UInt8 i;

    UInt16 crc = 0xffff; //seed
    for ( j = 0; j < len; j++)
    {
        UInt8 b = buf[j];
        for ( i = 0; i < 8; i++)
        {
            crc = ((b ^ (UInt8)crc) & 1) ? ((crc >> 1) ^ 0xA001) : (crc >> 1);
            b >>= 1;
        }
    }
    return crc;
}

```

New Member

Total Posts : 8  
Reward points : 0  
Joined: 5/13/2011  
Location: 0  
Status: **offline**

I made a little bit modification to the source code posted by "flamewatcher" in order to adapt to the 16bit dsPIC33. The C source code is listed below and reimplemented with assembly language in my project. In the source file "sie.s", the code below the label "\_\_CRC16:" calculates the CRC code and simultaneously copy the data into buffer "\_\_datay".

```

unsigned short _ComputeCRC16(const unsigned short * buf, int len)
{
    int j;
    unsigned char i;

```

---

```

    unsigned short crc = 0xffff; //seed

    for ( j = 0; j < len; j++)
    {
        unsigned short b = buf[j];
        for ( i = 0; i < 16; i++)
        {
            crc = ((b ^ (unsigned short)crc) & 1) ? ((crc >> 1) ^ 0xA001) : (crc >> 1);
            b >>= 1;
        }
    }
    return crc;
}

/*-----*/
/*- <14>  Hints about Module usb.c                      -*/
/*-----I'm NOT gorgeous dividing line-----*/

```

It's allowed to connect multiple devices with a HUB on the USB bus. Different type of device has different demand of transmission. For example, the size of data transmitted from a keyboard is very small but MUST be transmitted opportunely when a key is pressed. Because the data transmission is always launched by the host but not the device, the keyboard requires the host to launch transmission automatically in a specific interval. Another example is USB sound card. A stereo music which is played at 44.1KHz sample rate needs 44100x2x2 bytes per second. It's not too much and the host MUST send enough bytes within one second to avoid clicking. An USB mass storage device requires the host to send data as much as possible within one frame. Because of the limitation of bandwidth there must be a strategy to assign proper bandwidth for different device. Please refer to the section 4.7 on page 34 and section 5.4 - 5.8 on page 20 - 34 of USB2.0 Specification.

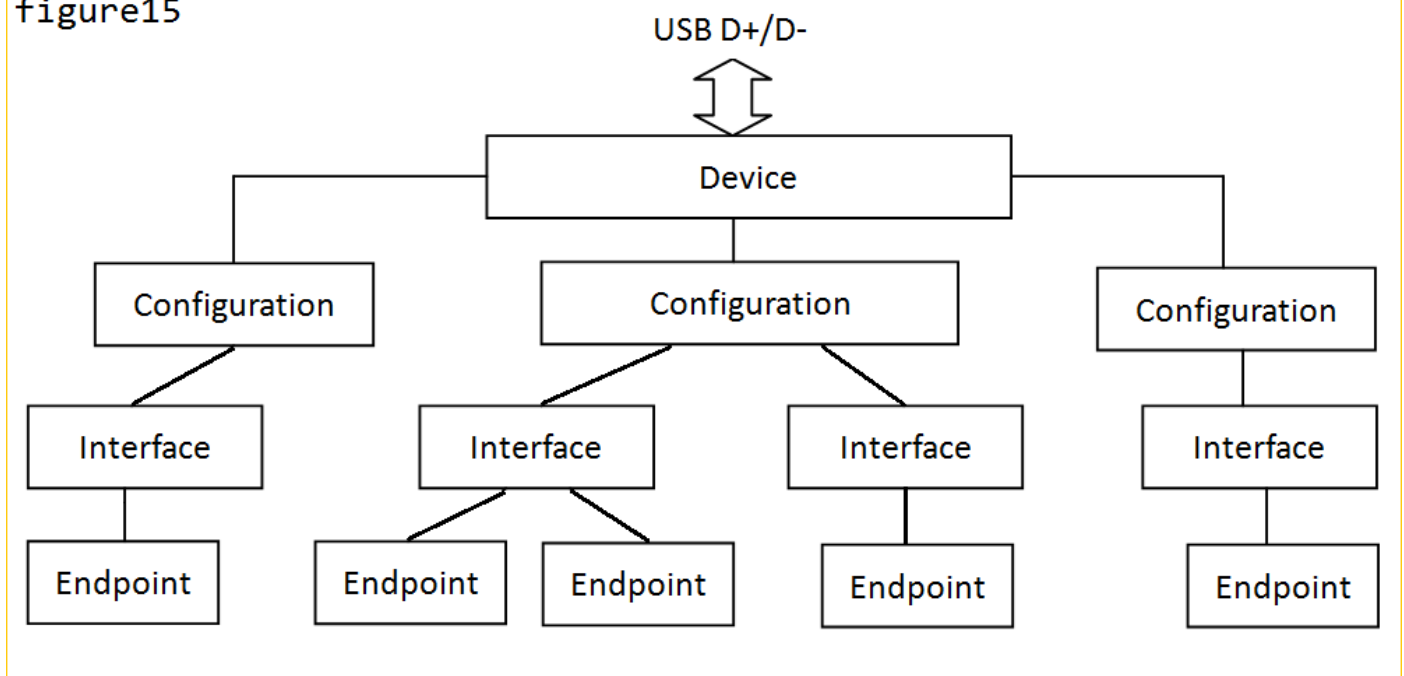
There are four types of transmission defined by USB2.0 Specification. The first one is "Bulk Transmission". It will be arranged into every frame as much as possible. USB mass storage devices should use this type of transmission. The second one is "Isochronous Transmission". A prenegotiated amount of USB bandwidth will be occupied for it with a prenegotiated delivery latency. It's used for USB audio/video devices. The third one is "Interrupt Transmission". It will be launched in a prenegotiated interval by the host. It's proper for USB mouse and keyboard. The last one is "Control Transmission". It's used to configure a device or transmit small amount of data. Minimum bandwidth is allocated for this type of transmission. Please refer to the section 4.7 on page 20 and section 8.5.2 to 8.5.5 on page 221 of USB2.0 Specification.

The devices declare the types of transmission that they need by reporting the "endpoints" they own to the host. An "endpoint" is represented by a 4bit digit which is called "endpoint address" and assigned a memory buffer. Assume that there is a device which needs to transmit data with "Bulk Transmission". It should allocate a memory buffer with 64 bytes length and report an endpoint address to the host. It MUST report some extra information about this endpoint address including "type of transmission" and "transmission direction" and "buffer length" etc. The host will allocate enough bandwidth for this endpoint according to this information. We should notice that every device MUST have an endpoint with address ZERO and only the "Control transmission" is allowed through this endpoint. All other endpoints are optional. Please refer to the section 5.3.1 on page 33 of USB2.0 Specification.

There are some devices which have multiple functions. For example, a keyboard combined with a touch panel is a composite device of keyboard and mouse. It has multiple endpoints to be assigned to the two functions. The keyboard function uses endpoint 2 & 3 and the mouse function uses endpoint 4 & 5. The endpoint 2 & 3 makes an "interface" for the keyboard and endpoint 4 & 5 makes another interface for the mouse. The interfaces can be merged into a group which is

called a "Configuration". The host activates a configuration and transmits data through endpoints which belongs to an interface when the device is plugged into the USB port. A complete device is formed with a bunch of configuration and its interfaces.

figure15



"Device/Configurations/Interfaces/Endpoints" are defined by a data structure which is called "Descriptor". There must be a "Device" descriptor built in a device with single or multiple "Configuration/Interface/Endpoint" descriptors. When the device is plugged into the host or hub, the device descriptor will be fetched first then the configuration/interface/endpoint descriptor.

In the "Device" and "Interface" descriptor there is a member which is called "Class". Some USB devices have predefined function such as keyboard and mouse and USB disc etc. When we design these types of device we can declare a predefined class code in descriptors. The host will know how to communicate with these devices and we don't have to build a driver for the host to support the device.

In most of device the class code in the device descriptor is usually zero and device class is defined by the interface descriptor. Some devices have their class code defined in device descriptor but it is not common. All predefined class codes are listed on <https://www.usb.org/defined-class-codes>. There is a specific descriptor for each predefined device class. It is inserted between the interface descriptor and the endpoint descriptor.

My project is a "HID" (Human Interface Device) device. There are some "subclass" predefined under this class and "protocol" predefined under a "subclass". I didn't use any "subclass" and "protocol". That means my project is a custom HID device. Basic data communication could be supported by the host but the host doesn't concern the "meaning" of these data.

The communication between the host and HID device is organized into a bunch of "Report". A "Report" is a bit stream which is split into small data elements. The bit width of a data element and the usage and value scope are defined by the device in a "Report Descriptor". Please refer to the section 5.2 - 5.6 on page 14 and section 6.2.2 on page 23 of HID1.11 Specification.

When a device is plugged into the host or HUB, the initial device address is 0. The host fetches all descriptors through a group of "Control Transfer" which are sent to "endpoint 0" and "address 0". The first token "SETUP" and the data is usually "0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00". (Please refer to the section 9.3 and 9.4 on page 248 of USB2.0

Specification.) The first two bytes is a command which means "Get Descriptor". The next two bytes indicates that a "Device Descriptor" will be fetched. The last two bytes is the byte length required by the host. In this example, the last two bytes is "0x40 0x00". That means the host wants to get "0x0040" bytes from the device. Following the token "SETUP" there are series of token "IN". Because the data attached to token "SETUP" is transmitted via "DATA0", the data of the first token "IN" has to be transmitted via "DATA1". The device has to split all data into small pieces according to the buffer length of endpoint 0. The host will repeatedly send token "IN" and the data PID is alternately switched between "DATA0" and "DATA1" until all data are sent by the device. At the end of the transmission the host sends a token "OUT" and a "ZLP" (zero length data attachment) with PID "DATA1" to the device. With that "ZLP" the complete data interaction is end up.

A critical problem is how the host knows all data has been sent by the device. The strategy is listed below.

- a. The host submits a maximum length to the device. Such as "0x0040" quoted above.
- b. If the real data length in device is longer than the maximum value submitted by the host, the device just sends part of data that fulfill the requirement of the host. Data remainder has to be discarded. Then the transmission will be ended automatically by the host.
- c. if the real data length in device is shorter than the maximum value submitted by the host, the device should send all data to the host. If the length of the last data block equals the buffer length of endpoint, An extra zero length data packet has to be sent to the host. The transmission will be ended when the host receives the "ZLP". The PID of this "ZLP" might be "DATA0" or "DATA1" and following the previous data packet.

This transmission consisted with a "SETUP" and a series of "IN" and a "OUT" with a "ZLP" as "DATA1" is called "Control Read" sequence. Another similar sequence is "Control Write", which consists with a "SETUP" and a series of "OUT" and a "IN" with a "ZLP" as "DATA1". The "Control Write" sequence is a little bit simpler because the host knows the real data length and will generate proper numbers of token "OUT". The device should send a "ZLP" as "DATA1" to the host when all data has been received.

The data stage (a series of "IN" or "OUT") is optional. Sometimes there isn't any data need to be sent. For example, when a new device address is allocated by the host, the host will send a token "SETUP" with 8 bytes data "0x00 0x05 0x03 0x00 0x00 0x00 0x00 0x00" to the initial device address 0. The first two bytes means "Set Address" and the third byte "0x03" is the value of new address. The last two bytes is "0x0000" that means there isn't any data sent to the device. The device just sends a "ZLP" as "DATA1" to complete this transmission.

There is a tiny trick in this interaction. The host sends a token "IN" at the end of transmission to obtain the "ZLP" as "DATA1". The device address in this token is still the initial value zero. The device MUST receive this token "IN" first and send the "ZLP" to the host then renew the device address when an "ACK" is sent by the host.

```
/*-----*/
/* <15>  API functions in Module usb.c                      -*/
/*-----I'm NOT gorgeous dividing line-----*/
```

In the source file "usb.c" there is a group of descriptors and four API functions. All three types of HID reports (Input/Output/Feature) are activated by the "HID\_ReportDescriptor" and all of them are 64 bytes length. I set two strings index in the "USB\_DeviceDescriptor" which are used to index the "Manufacture String" and "Product String". These two strings will be displayed on some host platform when the device is plugged.

The four API functions prototype are listed below:

- void USB\_vInit(void); /\* Power On Initialization \*/



It's not implemented so far.

- `BYTE USB_bRxRequest(void* Request); /* Fetch eight bytes data attached to token SETUP  
and process some standard requests */`

The pointer parameter "Request" is NOT allowed to be NULL. Only three standard requests are handled by this function -- Get Descriptor (Request[1] == 0x06), Set Device Address (Request[1] == 0x05) and Set Configuration (Request[0] == 0x00 && Request[1] == 0x09). Two standard requests defined in HID1.11 Specification -- Set Report (Request[0] == 0x21 && Request[1] == 0x09) and Get Report (Request[0] == 0xA1 && Request[1] == 0x01) are NOT handled and directly return "USB\_REQ\_SETUP" to the caller. An extra request -- Set Idle (Request[0] == 0x21 && Request[1] == 0x0A) -- which might be sent on some LINUX platform, is NOT handled and return "USB\_REQ\_SETUP" as well.

- `BYTE USB_bGetCtrlData(BYTE * dat, WORD siz, WORD exLength); /* Fetch all data attachment  
of token OUT after SETUP */`

The pointer parameter "dat" indicates the buffer and is NOT allowed to be NULL. The parameter "siz" indicates the byte length of the buffer. The third parameter "exLength" indicates the expected length defined in the data attachment of token SETUP. When the function "USB\_bRxRequest" is invoked, the "exLength" is represented by "Request[6]" and "Request[7]".

When the parameter "siz" is equal or greater than the parameter "exLength", all data attached to token OUT will be handled and a "ZLP" will be sent to the host by this function. When the parameter "siz" is less than "exLength", the data will not be handled correctly but no buffer overflow error generated.

The return value is the total bytes received from the host.

- `BYTE USB_bSendCtrlData(BYTE* dat, WORD siz, WORD exLength); /* Send data to the host via Control  
Read sequence */`

The pointer parameter "dat" points to the buffer in which the data is stored. It is allowed to be NULL. When "dat" equals NULL, the parameters "siz" and "exLength" MUST be set to zero simultaneously. The parameter "siz" is allowed to be zero and "exLength" MUST be zero simultaneously. The parameter "exLength" is the expected length defined in the data attachment of token "SETUP". When the "siz" and "exLength" are all zero, a "ZLP" as "DATA1" will be sent to the host. When "siz" is less than "exLength" and equals times that of "MAX\_PACKET\_SIZE", an extra "ZLP" will be sent to the host as the tail block of data. After all data sending, the "ZLP" as "DATA1" from the host will be waiting for. The return value is the length of the last data block.

```
/*-----*/
/*- <16>  Hints about Module hid.c                      -*/
/*-----I'm NOT gorgeous dividing line-----*/
```

There are two ways to send a report to a HID device on WINDOWS platform. First, invoke the WINDOWS API function "HidD\_SetFeature" to send a "Feature Report" to the device through a "Control Write" sequence. Second, invoke the other WINDOWS API function "HidD\_SetOutputReport" to send an "Output Report" to the device through the "Interrupt Transmission". If the device doesn't have "Interrupt Endpoint", the "Output Report" will be sent through a "Control Write" sequence as well. The device will receive 8 bytes data attachment of token "SETUP" as "0x21 0x09 0x00 0x03 0x00 0x00 0x40 0x00" when the function "HidD\_SetFeature" is invoked. If the function "HidD\_SetOutputReport" is invoked, the 8 bytes should be "0x21 0x09 0x00 0x02 0x00 0x00 0x40 0x00". For more details of these WINDOWS API please refer to <https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/hidsdi/>.

There are also two ways to receive reports from the device. We can invoke the WINDOWS API function "HidD\_GetFeature" to receive a "Feature Report" from a HID device. When this function is invoked a token "SETUP" and data attachment "0xA1 0x01 0x00 0x03 0x00 0x00 0x40 0x00" will be sent to the device. Another WINDOWS API function "HidD\_GetInputReport" can be invoked to read "Input Report" from a HID device through the "Interrupt Transmission". If there isn't any interrupt endpoint in the device, this function will read data via a "Control Read" sequence and the data packet of token "SETUP" is "0xA1 0x01 0x00 0x01 0x00 0x00 0x40 0x00". Please refer to the section 7.2 on page 50 of HID1.11 Specification.

```
/*-----*/
/*- <17>  API functions in Module hid.c                                -*/
/*-----I'm NOT gorgeous dividing line-----*/
```

If the host invokes "HidD\_SetFeature" to send 64 bytes to the device, the bitwise complement of all bytes will be calculated and stored in a buffer. These bytes will be sent back to the host when the function "HidD\_GetFeature" is invoked by the host. The first two bytes sent by the host are used as a counter in a loop which simulates a simple task launched by the device. After the loop ending the device will process the token "SETUP" generated by the function "HidD\_GetFeature". That means if the host sends some random bytes to the device, the firmware running inside the device will delay for a random period then send back the bitwise complements. If the host calls "HidD\_SetOutputReport" to send data to the device, all data will be send back as their original form without any delay.

There is a global variable "State" defined in the module "hid.c". The value of "State" is "COMMAND" when the data from the host is received and "RESPONSE" when the data sent back is ready. If the host invokes "HidD\_GetFeature" to read data from the device when the value of "State" is "COMMAND" still, a "ZLP" will be sent back to the host. We can return some specific data to the host to indicate that the real data isn't ready so the host is allowed to poll the data in a loop. Unfortunately it doesn't work because the module "sie.s" can NOT generate an interrupt when a token "SETUP" is received. In another words, the processing of token "SETUP" in the module "hid.c" is NOT in real time.

```
- void HID_vInit(BYTE Mode); /* Initialization when power on and chip reset */
```

The parameter "Mode" is used to indicate the reason of reset. It's "power on reset" or "software reset" or "watch dog reset" etc.

```
- BYTE HID_bRxRequest(void * Req, WORD siz); /* Handle some requests defined in HID1.11 Specification */
```

The pointer parameter "Req" points to a buffer with byte length indicated by the parameter "siz". If the "Req" is valid and "siz" equals 2, the first two bytes received from the host will be put into the buffer and the return value is "1" that means the data in the buffer "Req" is valid. If the return value is "0" that means there isn't any valid data returned through the buffer "Req".

```
- BYTE HID_bTxResult(void * dat, WORD siz); /* Return a data packet to the host */
```

The parameter "dat" points a buffer in which the data packet is stored. The parameter "siz" is the byte length of the data packet. So far this function is only used to set the value of global variable "State" and return "1" to the caller directly. All data interaction is implemented within the module "hid.c".

```
/*-----*/
/*- <18>  main.c                                                        -*/
/*-----I'm NOT gorgeous dividing line-----*/
```

The module "main.c" is very simple. The function "setup" is called by the module "sie.s" when the chip is powered on and reset. The function "loop" is the major task loop in that there is a short type variable "Req" which will be used as a



counter in a loop. The value of "Req" is set with the first two bytes of data received from the host when the function "HID\_bRxRequest" is called. The LED is lighted up before the loop and turned off when the loop starts. We will know the period of the loop through the state of the LED.

## Known BUG

When the device is plugged into an USB HUB, the communication fails occasionally when the host sends data to the device. The frequency of failure is related to the data sent by the host. Specifically if the host sends random data to the device repeatedly, the frequency of failure is very low. If the host sends all bytes with same value, such as 64 bytes 0xFF, the frequency of failure is higher. This bug is triggered only when the device is connected to a HUB. It's never been triggered when the device is connected to the host directly.