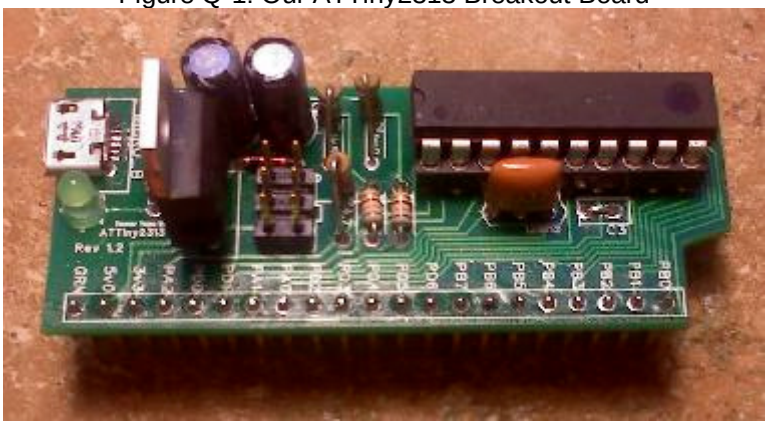
Appendix Q - Building a Simple USB Device

In this chapter, we will build a small USB device, learn how to build it using the Atmel® ATTINY2313[1] micro-controller, program the controller to accept USB commands, then communicate with our newly built device using the information you learned within the chapters of this book.

All USB devices have a common requirement, the USB communications interface. That is the hardware that is used to send and receive USB data. Therefore, the first thing we will do is create this common requirement using a few electronic components and solder them to a PCB with 21 pins that you can insert into a breadboard ready to build your USB device.

Figure Q-1: Our ATTiny2313 Breakout Board



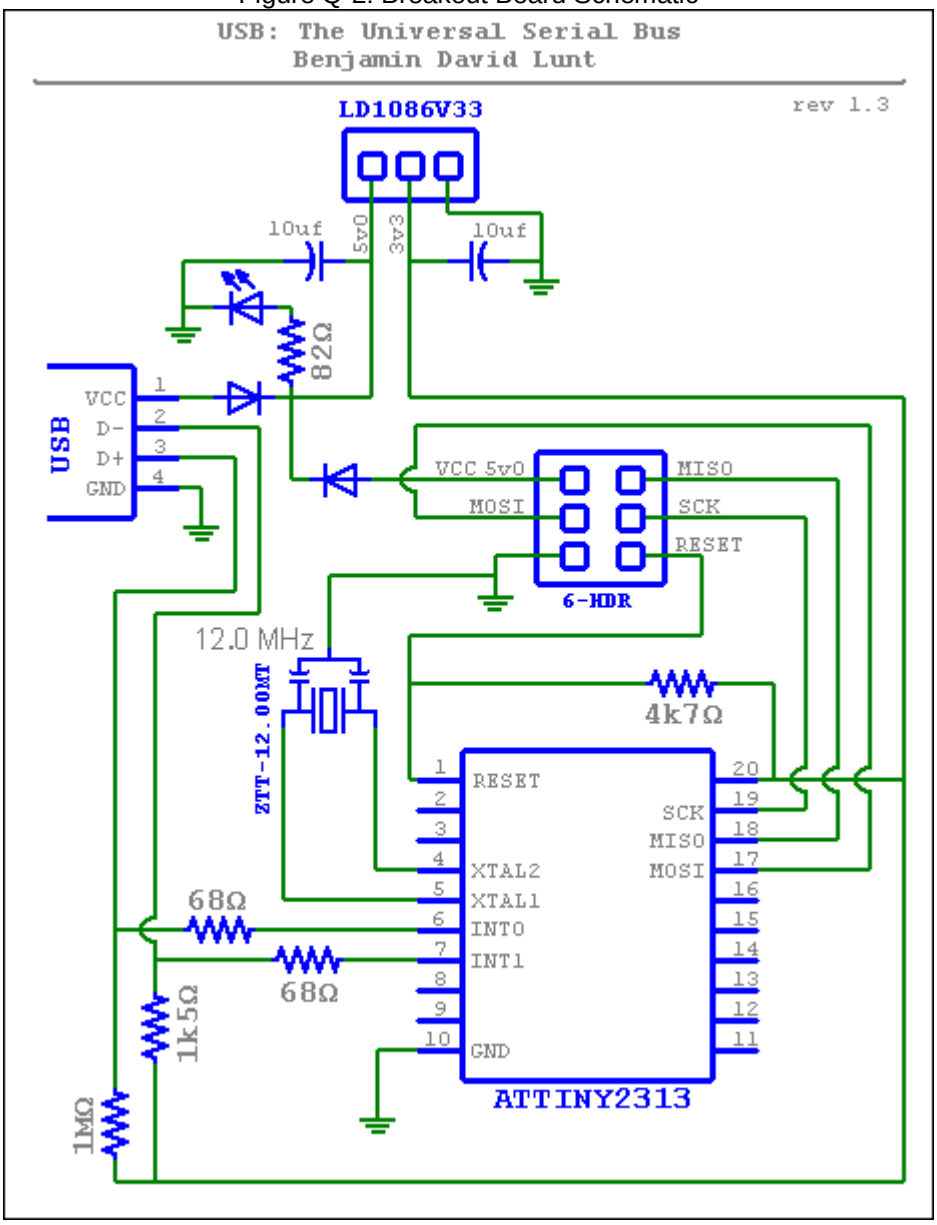
In the figure above, we have the USB connector on the left, a green LED and resistor to indicate that we have power from the USB connector, the 6 pin programming header, two diodes, the voltage regulator and its two capacitors, five resistors, a 12-Mhz resonator, and the ATTiny2312 micro-controller that does all the work. Then at the bottom of the PCB there are 21 pins that allow us to place these into a breadboard allowing us to build USB devices.

Included on the disc, in the misc folder, is the file I used with FreePCB to create the PCB above.

Throughout the remaining sections of this chapter, I will explain what you will need to build the board, how to build and program it, then how to use it to create your own USB devices.

[1] http://www.atmel.com/

Figure Q-2: Breakout Board Schematic



Our ATTiny2313 Breakout Board

Our breakout board uses a USB connector, a voltage regulator, the micro-controller, and a 12-Mhz resonator, along with a few other components used with each of these items. Figure Q-2, on the previous page is the schematic of the breakout board.

The following table shows the list of the components used. Please note that the price of the components was taken at time of this writing and does not include any applicable taxes or shipping.

Table Q-1: Breakout Board Components

	Part Number[1]	Description	Cost
1	USBTinyISP	The Programmer (www.fysnet.net/blog/2012/12/)	15.62
1	Breakout Board	Our PCB (fys@fysnet.net)	7.50
1	538-105017-0001	USB Type B Micro Connector	1.14
1	859-LTL-1CHG	Green LED	0.08
1	791-RC1/4-820JB	82 Ohm 1/4w resistor[2]	0.29
1	511-LD1086V33	Voltage Regulator	0.95
2	667-ECA-1HHG100	10 uF Capacitor	0.35
1	538-10-89-7062	6 pin (3x2) program header	0.67
2	78-BZX55C5V1-TAP	5v1 zener diode	0.07
1	791-RC1/4-105JB	1M Ohm 1/4w resistor[2]	0.29
1	791-RC1/4-152JB	1k5 Ohm 1/4w resistor[2]	0.29
1	791-RC1/4-472JB	4k7 Ohm 1/4w resistor[2]	0.29
2	660-CFS1/4CT52R680J	68 Ohm 1/4w resistor	0.15
1	575-199320	Optional 20 pin Socket	0.69
1	556-ATTINY2313A-PU	ATTiny2313 micro-controller	1.31
1	520-ZTT1200MT	12Mhz Resonator	0.50
1	649-68004-236	36 pin (1x36) header	0.83
		Total in USD	31.59

[1]All but the top 2 items are from www.mouser.com

[2]Resistor is not the one shown in Figure Q-1 but works just the same.

When you acquire these items, I recommend that you acquire multiple quantities of all but the top two items, especially the LED's and then even different colored LED's. You will need a few of them in the examples later in this chapter.

The first component needed for any USB device is of course the USB connector. Here I use a USB Type B Micro connector. It is a surface mounted connector and needs to be soldered in numerous places to the PCB so that it does not come loose, since it may have a cable attached and unattached numerous times.

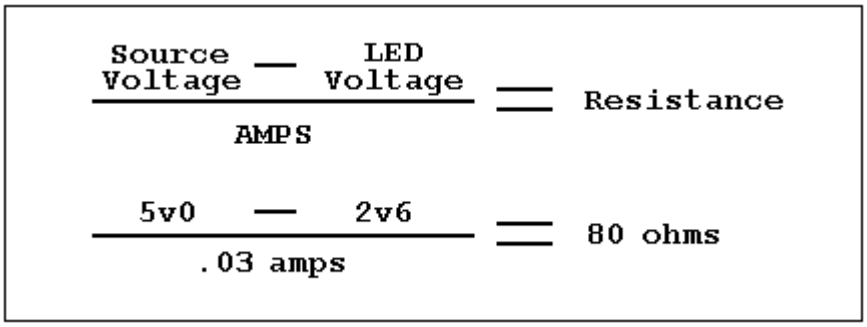
In a previous revision of this breakout board, I used a standard Type B connector, however, it used up a lot of room on the board. By using the Micro connector, I was able to free up some space and place an LED indicator next to it.

The next component is the green LED used to indicate that we have current at the connector. Since the USB provides about 5.0 volts, we must use a voltage resistor with the LED so that the 5v0 will not burn out the 2v6 LED.

In the electronics world, when a voltage, ohm reading, or other value is listed, the 'v' in voltage is used for the point in 5.0 volts. For example, 5.0 volts would be listed as 5v0, while 4.5k ohms would be listed at 4k5 ohms. Therefore, the 2v6 above indicates 2.6 volts.

A resistor will limit the amount of current to the LED, hence not allowing the LED to burn out. To calculate the amount of ohms needed for the resistor and LED pair, you take the source voltage of 5v0 and subtract the amount of voltage the LED requires, 2v6, and then divide by the amount of amps the LED will use. This will give you the resistance needed.

Figure Q-3: LED Resistance Calculator



With the LED I use here, it requires an 80-ohm resistor. Since it does not have to be exact, and common ohm values are much cheaper and more available, I rounded to an 82-ohm resistor. If you use a different LED than I do, make sure to calculate the correct resistance you will need for the resistor.

Since the USB provides 5v0 and the USB data lines for the micro-controller requires 3v3 volts, we need to use a voltage regulator to drop the voltage to the required 3v3. I used the Low Dropout LD1086, but any voltage regulator that outputs 3v3 will do. The specification for this regulator requires a 10uF capacitor on the 5v0 input as well as the 3v3 output.

You can get away without using the voltage regulator by using a couple of 3v6 diodes and a resistor on the data lines. However, this would not give us the two voltage sources for our breadboard.

The next component is the 6-pin programming header used by the programmer to program the micro-controller. See later in this chapter for more on programming the controller.

If you have the breakout board plugged in to a USB port, you should never have the programmer plugged in to the board, and visa-versa. However, just to be sure, I have placed two diodes on the board to not allow one voltage source to contaminate the other source if you accidentally have both items plugged in at the same time.

The next components are five resistors. Two are used as pull-up resistors for the two USB data lines, two more on the data lines themselves, while the fifth is used to pull up the Reset pin of the micro-controller.

With this project, I put a 1k5 Ohm resistor from 3v3 to the D- data line. This indicates to the hub that this device is a Low-speed device. Put the 1k5 Ohm on the D+ data line to indicate a Full-speed device, swapping the 1m0 Ohm resister to the D- side.

The next component is the 12-Mhz Resonator used by the micro-controller.

The last component is the micro-controller itself, the ATTiny2313. We will use two pins as data lines for the USB interface and eight I/O pins to control our USB device. See later in this chapter for how we do this. Next, soldering the components.

Soldering the Components

To solder the components, you will need a 25-Watt soldering iron with a small tip. The tip should be small enough not to get in the way of other pins, but large enough to hold good heat. The tip should be clean and tinned before and after use.

The term “tinned” means that it is covered with solder. This is to keep the iron’s tip from corrosion.

You should have a good solder paste. This will clean and flux the components once heat is applied. I use Oatey’s *No. 5 Lead Free Solder Paste* No. 30011. It is good practice to always clean the residue paste from the component once it has cooled. You can clean it with denatured alcohol and a small flat paint brush.

When soldering, apply a small amount of paste to the pin and solder pad of the PCB. Place a little dab of solder on the tip of your warmed iron, then place the tip of the iron on the pin and solder pad. Once the pin and solder pad are warm enough, the solder will flow from the iron to the pin and pad. Hold the iron on the pin and pad for a split second longer to be sure the solder takes, then remove the iron. Continue with each component with this technique, snipping the excess lead lengths from the PCB once the solder has cooled. Be sure to use sharp cutters to cut the leads. Dull cutters may "shock" the lead, breaking the connection from the solder.

Another solder technique is to hold the iron on the pin and pad, then applying solder to the other side of the pad. Once the pad and pin are warm enough, the solder will flow toward the iron. This is considered a "better" technique. However, this uses both hands to solder where the first technique can be done with one hand holding the project.

Once you have soldered all the components to the board, finally solder the 21-pin header to the backside so that the pins point down. I left the drill holes a little large in the PCB for this header so that you can solder it at a slight angle forward. This will "lift" the PCB from the breadboard a little when inserted.

The Programmer

The programmer I use is a stripped down version of the USBtinyISP AVR Programmer Kit (USB SpokePOV Dongle) - v2.0 from Adafruit Industries, https://learn.adafruit.com/usbtinyisp that I made myself. Any 6-pin ISP programmer designed for the ATTiny series will work.

I use the Windows® version of the driver located at the URL above, by clicking on the *Download* link and downloading the *Windows USBtinyISP* driver. Unzip the file to an empty folder, then plug in the programmer. When prompted, point the Windows' "Found New Hardware" installer to this folder. When the install is complete, unplug the programmer then plug it back in to be able to use it.

The Cross Compiler

Next, you will need an AVR cross compiler. I use the one at http://winavr.sourceforge.net/. Install the compiler by simply running the install file and allowing it to install to an empty folder. Make sure your PATH environment variable points to this WinAVR\bin folder.

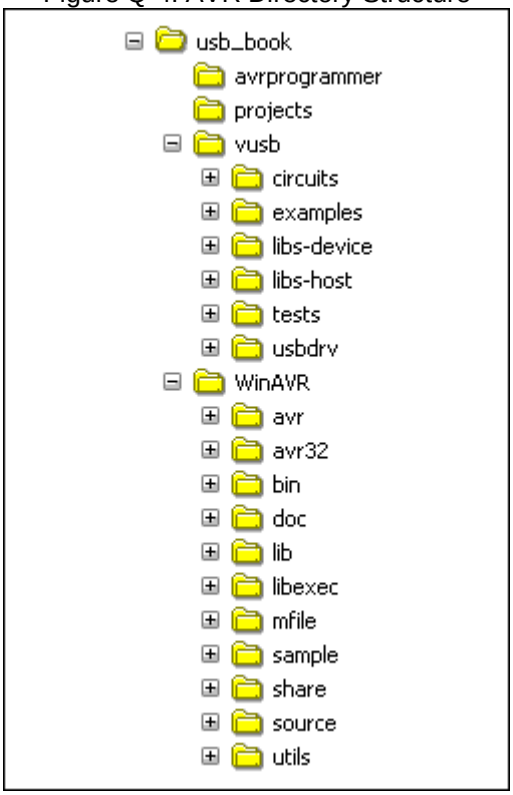
You will also need the USB-LIB code found at

http://www.obdev.at/products/vusb/download.html

This download contains the USBDRV folder that you will need for each device you build. Unzip it to the same group of folders.

You should now have a directory structure similar to the figure shown here.

Figure Q-4: AVR Directory Structure



The *avrprogrammer* folder is where you might have put the driver files for the programmer, the *vusb* folder is where you might have put the USB Library files, which includes the needed *usbdrv* folder, and the *WinAVR* folder is where you might have installed the Cross Compiler. The *Projects* folder is one that you might want to create to hold all of your projects.

Once you have the AVR compiler and the USB library installed, you can now start to write, compile, and flash code to the ATTiny2313.

Fusing and Flashing

One of the first things we will do is create a simple blinking LED app just so that we know that the programmer and the breakout board works, and that we soldered everything correctly.

In the electronics world, a blinking LED project is the same as the "*Hello, World*!" project in the programming world.

In the *projects* folder, create a folder called *blink*.

Now create a single source file called *main.c*.

Listing Q-1: Blink: main.c

```
#include <avr/io.h>
#include <avr/interrupt.h>  // for sei()

#include <util/delay.h>     // for _delay_ms()

int main(void) {

  // make bottom 2 PORTB pins output
  DDRB = 3;  // bits 0 and 1

  while (1) {        // main event loop
    PORTB = 1; // Turn LED on PB0 on and PB1 off
    _delay_ms(500);
    PORTB = 2; // Turn LED on PB0 off and PB1 on
    _delay_ms(500);
  }

  return 0;
}
```

You will now need a Makefile to be able to build and flash the project. Create a file called *Makefile* with the contents in the Listing below.

Listing Q-2: Blink: Makefile

```
CC = avr-gcc
OBJCOPY = avr-objcopy
DUDE = avrdude
RM = del

CFLAGS = -Wall -Os -Iusbdrv -mmcu=attiny2313 -DF_CPU=12000000
OBJFLAGS = -j .text -j .data -O ihex
```

```
DUDEFLAGS = -p attiny2313 -c usbtiny

# Object files for the firmware
OBJECTS = main.o
# By default, build the firmware, but do not flash
all: main.hex

# Flash the firmware by typing "make flash" on command-line
flash: main.hex
  $(DUDE) $(DUDEFLAGS) -v -U flash:w:$<

# This will set the FUSE bits of the chip
fuse:
  $(DUDE) $(DUDEFLAGS) -U lfuse:w:0xef:m

# Housekeeping if you want it
clean:
  $(RM) *.o *.hex *.elf

# From .elf file to .hex
%.hex: %.elf
  $(OBJCOPY) $(OBJFLAGS) $< $@

# Main.elf requires additional objects, not just main.o
main.elf: $(OBJECTS)
  $(CC) $(CFLAGS) $(OBJECTS) -o $@

# From C source to .o object file
%.o: %.c
  $(CC) $(CFLAGS) -c $< -o $@

# From assembler source to .o object file
%.o: %.S
  $(CC) $(CFLAGS) -x assembler-with-cpp -c $< -o $@
```

Now by typing MAKE at the DOS command line, you should be able to make the firmware file *main.hex*.

Make sure you use the MAKE.EXE from the WinAVR install. If you have other compilers installed and your PATH environment variable has a pointer to one of them first, it may try to use that MAKE.EXE. This may produce errors.

Now you need to set the fuse bits of the micro-controller. This only needs to be done once per project and is explained on the next page.

The ATTiny2313 has an internal Oscillator (crystal) that is set by default to 1-Mhz, less than the 12-Mhz we desire. Therefore, by setting the CKSEL bits (bits 3:0) to any value of 1000 to 1111, we tell the micro-controller to use our external 12-Mhz clock attached to XTAL1 and XTAL2. We also set the SUT0 and SUT1 bits, and the CKOUT bit, but clear the CKDIV8 bit. Remember that clearing a bit indicates that fuse is set. See page 160, Table 68 of the ATTiny2313 specification.

Make sure you have the programmer plugged into your host and installed correctly. Now plug the programmer's 6-pin cable to the programming header of the breakout board. Make sure that the red line on the ribbon is on the side with the circle on the breakout board. i.e.: The ribbon should be lying over the regulator, not the resistors (see Figure Q-5). The red line side should be near the two capacitors. If the green LED on the breakout board is not lit, something is wrong.

At the C:\ prompt enter the following:

```
C:\>MAKE FUSE<enter>
```

This will write 0xEF to the low fuse byte. If everything is working correctly, you should have not seen any errors displayed. If you saw any errors displayed, double check all your connections and solder joints and try again.

Now you can flash the firmware to the micro-controller.

At the C:\ prompt enter the following:

```
C:\>MAKE FLASH<enter>
```

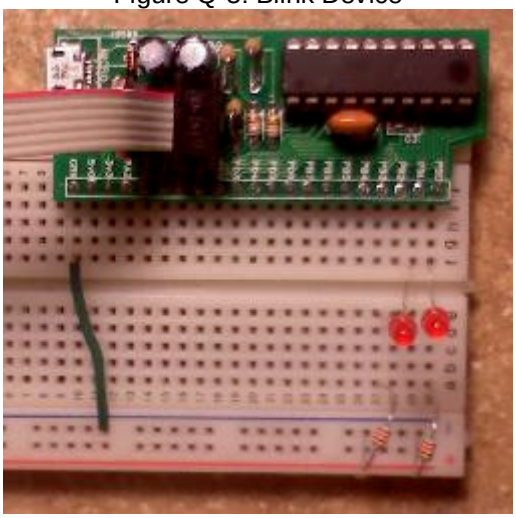
Again, watch for any errors that might be displayed. If no errors were found, you are ready to build the *Blink* device.

Build Blink Device

To build the *Blink* device, simply plug the breakout board into your breadboard so that you can plug an LED lead into the PB0 and another into the PB1 pin groups. Then put the other lead of the LED's into an empty group. Place a resistor between this group of pins and the GRN pin group on the breadboard. Be sure to connect the GRN pin to the GRN group as show in Figure Q-5 on the next page.

Remember that the longer lead on the LED is the positive side.

Figure Q-5: Blink Device



Note that the programming cable is still attached in the figure above. This is to show you which way it should be attached. You should always disconnect the programmer after programming the micro-controller.

Now plug in the breakout board to a USB power supply, either a charging unit or an actual USB port on a computer.

If you plug the device into a port on a computer, the host operating system will complain that you attached an unknown device. This is okay for now, we haven't gotten to the point of an actual USB device yet. We just want the 5v0 current it provides.

If you did everything correctly, the two LED's should flash alternating between the two about every half-second. If this is the case, we can move on to creating an actual USB device, sending and receiving data.

However, if you have not been able to get the LED's to flash, you need to go back and check all of your connections, solder joints, component orientation, program code, etc.

Troubleshooting

This section is incase you did not get the *Blink* project to work, listing a few questions to make sure you did everything correctly. The top of the next page shows a list.

- ✓ When you plugged in the programmer, did your Host accept the device? Was there an error message displayed? If you go to the Systems Menu, does the device show within the list of USB devices attached?
- ✓ When the programmer is plugged in to the Host, does the programmer's green LED light up? If not, there is no power to the programmer.
- ✓ When the programmer is plugged in to the Host and is plugged in to the breakout board, does the green LED on the breakout board light up? If not, is the ribbon coming from the regulator side?
- ✓ When you type MAKE FUSE on the command line, are there any errors on the screen due to the HOST not finding the AVR BIN directory?
- ✓ When you type MAKE FUSE on the command line, does the AVR_DUDE command show a "Check Connections" error?

If all check points above have been passed, and you still cannot get the programmer to program your breakout board, you will need to check all of the solder joints and components on the board, both the breakout board and, if you assembled the programmer, the programmer's board. Take an ohm tester and place one lead on one pin, and follow the traces on the board checking that there is continuity to each pin on that trace. Also make sure that all components are oriented correctly. The resonator and resistors can be either direction, but the diodes, LED, capacitors, regulator, and micro-controller must be oriented correctly.

Building an Actual USB Device

Fortunately, someone has already written the code we need for our firmware to be able to have our USB device communicate with the host. This code is in the *usbdrv* directory shown in Figure Q-4. Please read all applicable license documents in that folder.

Since we include that code with our projects, we don't have to worry about the actual USB communications. All we have to do is write code to use the functions it provides for us.

The first device we will build is a small example of applying power to the eight pins of the micro-controller, in turn lighting eight LED's. The second device will show how to read input from these eight pins. With this information, you should be able to create most any simple USB device, which would require current to run a motor or LED, or require input such as a sensor.

The attiny2313 micro-controller we are using actually has more than eight available pins. It has eight on Port B and seven on Port D. However, we are using four of the seven on Port D for communication and timing. Therefore, we can actually have a combination of eleven I/O pins.

Build *OUTPUT* Device

Create a directory called *output* in your *projects* folder. Copy the *usbdrv* folder from the *vusb* folder to this folder. This is the code required to have USB communication. Within this folder, rename the *usbconfig-prototype.h* file to *usbconfig.h*. This is the configuration file. In most cases, you will only need to modify this file, leaving the rest of the files within this folder unmodified.

For this project, make the following modifications to *usbconfig.h* starting with the "---- Hardware Config ----" section.

- `#define USB_CFG_IOPORTNAME D`
- `#define USB_CFG_DMINUS_BIT 3`
- `#define USB_CFG_DPLUS_BIT 2`

The above is telling the USB code to use Port D, pins PD3 and PD2 as the USB communications pins.

- `#define USB_CFG_CLOCK_KHZ (F_CPU/1000)`

Above defines F_CPU in the *Makefile* as 12 Mhz. (modification may not be needed)

- `#define USB_CFG_IMPLEMENT_FN_READ 0`
- `#define USB_CFG_IMPLEMENT_FN_WRITE 0`

The Host is not transferring data from the device other than the eight bytes within the SETUP request packet. If we were using control-in or control-out packets, other than the SETUP packet, we would need to set these to 1, as in the next example later in this chapter.

- `#define USB_CFG_VENDOR_ID 0xc0, 0x16`
- `#define USB_CFG_DEVICE_ID 0xdc, 0x05`
- `#define USB_CFG_DEVICE_VERSION 0x00, 0x01`
- `#define USB_CFG_VENDOR_NAME 'V', 'O', 'T', 'I'`
- `#define USB_CFG_VENDOR_NAME_LEN 4`
- `#define USB_CFG_DEVICE_NAME \`
  `'U', 'S', 'B', 'E', 'x', 'a', 'm', 'p', 'l', 'e'`
- `#define USB_CFG_DEVICE_NAME_LEN 10`

The above lines define the Vendor and Device ID's and string values used within the device's Device Descriptor. However, there are a few guidelines. Please see the *USB-IDs-for-free.txt* file for more information. If you will never distribute your device, you may use whatever values you wish.

- `#define USB_CFG_DEVICE_CLASS 0xFF`
- `#define USB_CFG_DEVICE_SUBCLASS 0`

The above defines are used for the Class and Subclass fields in the device's Device Descriptor. We will just use the Vendor Specific value for this example.

- `#define USB_CFG_INTERFACE_CLASS 0`
- `#define USB_CFG_INTERFACE_SUBCLASS 0`
- `#define USB_CFG_INTERFACE_PROTOCOL 0`

The above lines are used for the Class, Subclass, and Protocol fields in the device's Interface Descriptor.

- `/* #define USB_CFG_HID_REPORT_DESCRIPTOR_LENGTH 24 */`

Make sure this #define is commented out.

Any other values should be zero or left as is.

Now create the *main.c* source file with the contents listed in Listing Q-3 below.

Listing Q-3: Output: main.c

```
#include <avr/io.h>
#include <avr/wdt.h>
#include <avr/interrupt.h>  // for sei()

#include <util/delay.h>     // for _delay_ms()

#include "usbdrv.h"

// ------------------------------------------------------------------

// this gets called when custom control message is received
USB_PUBLIC uchar usbFunctionSetup(uchar data[8]) {
  usbRequest_t *rq = (void *) data;

  if ((rq->bRequest >= 0) && (rq->bRequest <= 7)) {
    PORTB ^= (1 << rq->bRequest); // toggle LED on pin
    return 0;
  }

  return 0;
}
```

```
int main(void) {
  wdt_enable(WDTO_1S);
  usbInit();

  // make all PORTB pins output
  DDRB = 0xFF;
  PORTB = 0;

  sei();
  while (1) {        // main event loop
    wdt_reset();
    usbPoll();
  }

  return 0;
}
```

The code in Listing Q-3 above simply catches all SETUP packets, and if the Request is zero through seven, toggles that respective bit in the PORTB register. If the bit is on, power is applied to that pin.

Now create the make file, *Makefile,* with the contents listed in Listing Q-4 below.

Listing Q-4: Output: Makefile

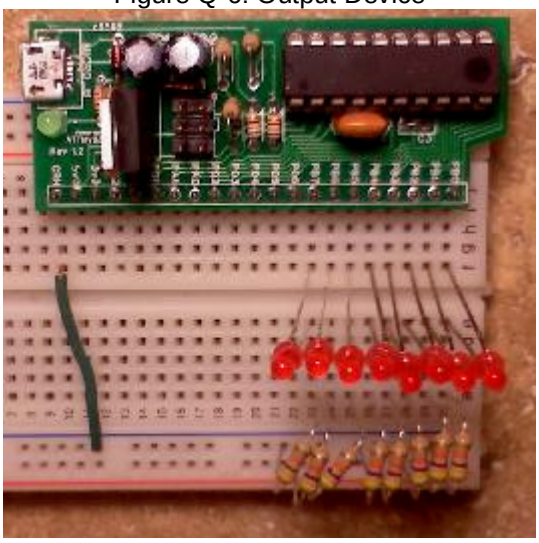
```
CC = avr-gcc
OBJCOPY = avr-objcopy
DUDE = avrdude
RM = del
CFLAGS = -Wall -Os -Iusbdrv -mmcu=attiny2313 -DF_CPU=12000000
OBJFLAGS = -j .text -j .data -O ihex
DUDEFLAGS = -p attiny2313 -c usbtiny

# Object files for the firmware
OBJECTS = usbdrv/usbdrv.o usbdrv/usbdrvasm.o main.o

# By default, build the firmware, but do not flash
all: main.hex

# With this, you can flash the firmware by just
#   typing "make flash" on command-line
flash: main.hex
  $(DUDE) $(DUDEFLAGS) -v -U flash:w:$<

# This will set the FUSE byte of the chip
fuse:
```

```
  $(DUDE) $(DUDEFLAGS) -U lfuse:w:0xef:m

# Housekeeping if you want it
clean:
  $(RM) *.o *.hex *.elf usbdrv\*.o

# From .elf file to .hex
%.hex: %.elf
  $(OBJCOPY) $(OBJFLAGS) $< $@

# Main.elf requires additional objects to the firmware,
#   not just main.o
main.elf: $(OBJECTS)
  $(CC) $(CFLAGS) $(OBJECTS) -o $@

# Without this dependence, .o files will not be recompiled
#  if you change the config file.
$(OBJECTS): usbdrv/usbconfig.h

# From C source to .o object file
%.o: %.c
  $(CC) $(CFLAGS) -c $< -o $@

# From assembler source to .o object file
%.o: %.S
  $(CC) $(CFLAGS) -x assembler-with-cpp -c $< -o $@
```

Notice that we now include the USB code in the *usbdrv* directory with our build.

Now create the device. Plug the breakout board into the bread-board as shown in Figure Q-6 on the next page, placing eight LED's in a row starting from the right with PB0, moving left, ending with PB7. Then place eight resistors, one lead in the same group for each respective LED with the other lead in the Ground group. Then place a single jumper from the GND pin on the breakout board to the Ground group on the breadboard.

Figure Q-6: Output Device



Connect the programmer to the Host and the breakout board.

At the C:\ prompt enter the following:

```
C:\>MAKE<enter>
```

If you didn't receive any errors, continue with:

```
C:\>MAKE FUSE<enter>
C:\>MAKE FLASH<enter>
```

Disconnect the programmer.

Now connect the breakout board to the host and simply send a Setup Request Packet, defined in Table Q-2 shown below, to the breakout board with a request value of zero through seven, turning on or off the respected pin's LED.

Table Q-2: Toggle Pin Request

Offset	Field	Size	Value	Description
0	Request type	1	**0xC0**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type 0 = Standard 1 = Class **2 = Vendor**

				3 = Reserved Bit 4:0 Recipient **0 = Device** 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0-7**	Pin (0, 1, 2, ..., 7)
2	Value	2	**0x0000**	High byte: 0x00 = not used Low byte: 0x00 = not used
4	Index	2	**0x00**	Interface number (0x00)
6	Length	2	**0x00**	Length

Build *INPUT* Device

Create a directory called *input* in your *projects* folder. Again, copy the *usbdrv* folder from the *vusb* folder to this folder and rename the *usbconfig-prototype.h* file to *usbconfig.h*.

Once you get more familiar with the way the code works and how the makefile's work, you can copy only the *usbconfig.h* file for each project, modifying the *makefile* slightly. Since the actual code within the *usbdrv* folder will be the same for every project, there is no need to copy it every time. Just be sure to re-compile the code for each project since the *usbconfig.h* file will change the #defines and how it is compiled.

For this project, make the same modifications you made for the *output* project before, with the following addition modifications.

- `#define USB_CFG_IMPLEMENT_FN_READ 1`
- `#define USB_CFG_IMPLEMENT_FN_WRITE 0`

The Host will be transferring data from the device.

- `#define USB_CFG_INTERFACE_CLASS 3`
- `#define USB_CFG_INTERFACE_SUBCLASS 0`
- `#define USB_CFG_INTERFACE_PROTOCOL 0`

The above lines are used for the Class, Subclass, and Protocol fields in the device's Interface Descriptor.

- `#define USB_CFG_HID_REPORT_DESCRIPTOR_LENGTH 24`

Uncomment the above line and make sure it is set to 24.

Any other values should be the same as the *output* project, zero, or left as is.

Now create the *main.c* source file with the contents listed in Listing Q-5 listed below.

Listing Q-5: Output: main.c

```
#include <avr/io.h>
#include <avr/wdt.h>
#include <avr/interrupt.h>  // for sei()

#include <util/delay.h>     // for _delay_ms()

#include <avr/eeprom.h>
#include <avr/pgmspace.h>   // required by usbdrv.h

#include "usbdrv.h"

static struct {
  uint8_t pins[8];
} state_report;

PROGMEM char usbHidReportDescriptor[42] = {
  0x06, 0x00, 0xFF,     // USAGE_PAGE (Generic Desktop)
  0x09, 0x01,           // USAGE (Vendor Usage 1)
  0xA1, 0x01,           // COLLECTION (Application)
  0x15, 0x00,           // LOGICAL_MINIMUM (0)
  0x26, 0xFF, 0x00,     // LOGICAL_MAXIMUM (255)
  0x75, 0x08,           // REPORT_SIZE (8)

  // read state
  0x85, 0x00,           // REPORT_ID (0)
  0x95, sizeof(state_report), // REPORT_COUNT
  0x09, 0x00,           // USAGE (Undefined)
  0xB2, 0x02, 0x01,     // FEATURE (Data,Var,Abs,Buf)

  0xC0                  // END_COLLECTION
};

// ------------------------------------------------------------------

#define CONFIG_TOP ((uint8_t *) 0)
#define CONFIG_LEN 128

static uchar currentAddress;
static uchar bytesRemaining;
static uchar report_id;
```

```
void buildStateReport() {
  uchar pins;

  pins = PINB;
  state_report.pins[0] = (pins & _BV(PB0)) ? 1 : 0;
  state_report.pins[1] = (pins & _BV(PB1)) ? 1 : 0;
  state_report.pins[2] = (pins & _BV(PB2)) ? 1 : 0;
  state_report.pins[3] = (pins & _BV(PB3)) ? 1 : 0;
  state_report.pins[4] = (pins & _BV(PB4)) ? 1 : 0;
  state_report.pins[5] = (pins & _BV(PB5)) ? 1 : 0;
  state_report.pins[6] = (pins & _BV(PB6)) ? 1 : 0;
  state_report.pins[7] = (pins & _BV(PB7)) ? 1 : 0;
}

uchar usbFunctionRead(uchar *data, uchar len) {
    if (len > bytesRemaining) len = bytesRemaining;
    eeprom_read_block(data, CONFIG_TOP + currentAddress, len);
    currentAddress += len;
    bytesRemaining -= len;
    return len;
}

usbMsgLen_t usbFunctionSetup(uchar data[8]) {
  usbRequest_t *rq = (void *)data;

  if ((rq->bmRequestType & USBRQ_TYPE_MASK) == USBRQ_TYPE_CLASS)
  {  // HID class request
    report_id = rq->wValue.bytes[0];
    if (rq->bRequest == USBRQ_HID_GET_REPORT) {
      if (report_id == 0) {
        buildStateReport();
        usbMsgPtr = (void *) &state_report;
        return sizeof(state_report);
      }
    }
  }

  return 0;
}

int main(void) {
  wdt_enable(WDTO_1S);
  usbInit();

  // make all PORTB pins input
  DDRB = 0;
```

```
  sei();
  while (1) {        // main event loop
    wdt_reset();
    usbPoll();
  }

  return 0;
}
```

The code in Listing Q-5 simply waits for a SETUP packet from the host, that sends the values in Table Q-3 below.

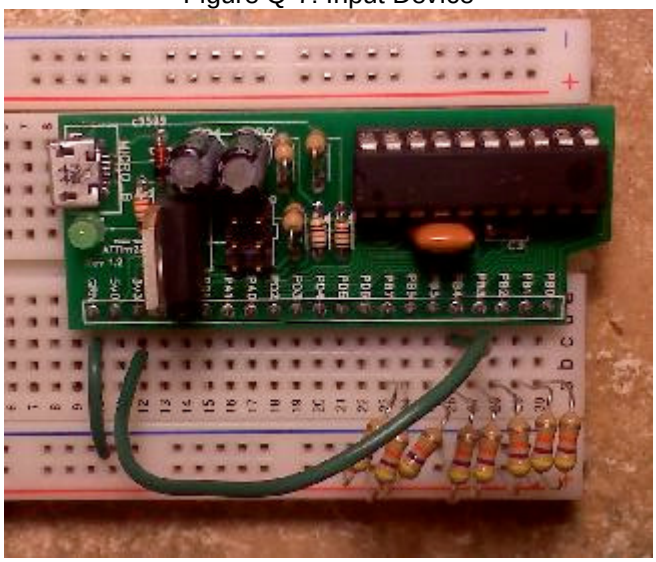
Table Q-3: Get Report Descriptor Request

Offset	Field	Size	Value	Description
0	Request type	1	**0xA0**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient **0 = Device** 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x01**	HID Get Report
2	Value	2	**0x0000**	High byte: 0x00 = not used Low byte: 0x00 = index 0
4	Index	2	**0x00**	Interface number (0x00)
6	Length	2	**0x08**	Descriptor length retrieve from HID Descriptor above.

The code will then create an 8-byte packet that holds a boolean value for each respective pin's input and place it at address zero of the chip. Then when the host sends the IN packet, it will transfer the requested amount of bytes, up to eight, to the Host.

Now copy the *Makefile* from the *Output* project folder to this folder. You should not have to make any modifications to it. Make the code and flash it to the breakout board. Then setup the breakout board and breadboard similar to Figure Q-7 on the next page, but without the wire connecting 3v3 and PB4 as shown, for now.

Figure Q-7: Input Device



Once you have connected the device to the host, send the request in Table Q-3. You should get an 8-byte array of all zeros. Now, take the long lead wire and connect the breakout board's 3v3 pin to PB4 on the breadboard, as shown in the figure above, and request the packet again. You should now get an array of seven zeros and a one. Try connecting the lead to other pins and requesting the packet again.

The eight resistors are needed to pull-down the eight pins. The Attiny2313 will have small voltages on the pins due to various other reasons, which will give false values in your 8-byte array packet. Therefore, if you use a 47k Ohm resistor to pull the pin to ground, you will eliminate this issue. Since the resistor has a high resistance, when you apply voltage to the pin group, very little voltage will be lost through the resistor to ground. I would use as large of a resistance as you can and still get good results. I have found that a 47k Ohm resistor works well.

Conclusion

With the information you now have, you should be able to use your imagination and create such things as a wheeled robot with proximity sensors, or other items such as an LED cube. You are only limited by your imagination.

Please note that some motors need additional circuits to control them correctly. Search for "stepper motor control" via your favorite web search tool.

Other Comments

Something that you might also want to look into is the ATTiny85 or similar chip. The ATTiny2313 used in this chapter does not have an ADC, Analog to Digital Converter, where as the ATTiny85 does. The ATTiny45 and ATTiny85 chips communicate via USB just as the ATTiny2313 used here with only a few modifications to the source code and Makefile, however they do not have as many I/O pins.

The specification for the ATTiny2313 used in this chapter can be found at.

http://www.atmel.com/Images/doc2543.pdf

I also want to thank the following people for their help in one way or another on this chapter.

Joonas Pihlajamaa	http://codeandlife.com/
Dan Stahlke	http://www.stahlke.org/dan/
Bernhard Schornak	http://thepoolofhumanity.blogspot.com/

You can find more information about this project at:

http://www.fysnet.net/attiny2313.htm
http://www.fysnet.net/the_universal_serial_bus.htm

Latest revision: 10 Oct 2014