*www.avrbeginners.net*

C/USB Tutorial

# *USB Control Transfers with LUFA*

Author: Christoph Redecker[1]
Version: 1.1.3

1 With lots of help from Dean Camera, the author of LUFA.

Getting to grips with USB is not easy. The USB specification is 650 threatening pages long. Fortunately, there is a layer of abstraction that mortals can actually understand: the USB device framework. It is described in chapter 9 of the USB specification.[2] In this tutorial, it is assumed that you have read and understood this very chapter. You can also have a look at other sites on the internet which try to explain USB: "USB in a NutShell"[3] at Beyond Logic does a good job. In fact, many diagrams in this document are redrawings of those found on that site.

2 You can download the USB specification at `http://www.usb.org`

3 `http://www.beyondlogic.org/usbnutshell/`

In the AVR world, LUFA[4] (Lightweight USB Framework for AVRs) takes care of all the USB related stuff up to the device framework level. At this level, we first need to talk about endpoint 0, which every USB device must have, and the overall structure of USB control transfers. LUFA offers some bits and pieces that we can use to process control requests. This is fairly easy to figure out once you've found the relevant functions in the LUFA documentation. But there is more to it, because Dean (the author of LUFA) has included some convenience functions. They can make the USB code a bit more readable.

4 `http://www.fourwalledcubicle.com/lufa.php`; the LUFA realease used in this tutorial is 120219.

An 8–channel RC servo controller will serve as a "real world" example. We will create a simple interface for it and call that from the USB code. However, the servo controlling code is *not* part of this tutorial.

## *Endpoint 0 and Control Transfers*

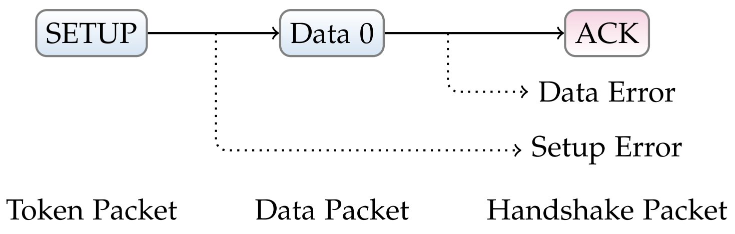
Endpoint 0 is the endpoint used for device enumeration. During enumeration, the host assigns an address to the device and reads device information, such as the device class. The host does so by sending control requests to the device, and the device will respond accordingly. LUFA takes care of all that almost automagically once we have defined the device and configuration descriptors. We could now lean back and say "So why do I need to understand it? LUFA does it for me!", but control requests can also be used for our own stuff.

### *The Setup Transaction*

Control transfers always start with a Setup transaction, followed by optional Data transactions, and are terminated with a Status transaction. The Setup transaction consists of the SETUP token (which contains the device address and the endpoint number), a data packet (which contains the setup packet, which in turn specifies the details of the control request) and, finally, the handshake.

The device can acknowledge the request by sending an ACK packet to the host in the handshake transaction. It is not allowed for the device to send a STALL or NAK packet. The following figure summarises the setup transaction:

In USB terms, the "device" is called "function". I'm including this note here to make clear that I'm not very serious about using correct USB terms.



Indication of transfer direction:

Host-to-Device

Device-to-Host

Errors in the setup token or data packet are indicated by the absence of an ACK sent by the device.

When a SETUP token has been received by the AVR USB controller, the RXSTPI flag in UEINTX is set. LUFA will recognise that and fire `EVENT_USB_Device_ControlRequest()`. This function can be implemented by the user to catch control requests. The user can choose to

There are many more events in LUFA. See Modules→USB Core→USB Events.

- do nothing about the request and leave the processing to LUFA,
- process the request, but let LUFA do its own processing as well,
- process the request and signal to LUFA that the request must not be touched any more.

When the control request event is fired, LUFA has already received the data packet. We do not need to worry about that. The information contained in the data packet is the setup packet, which details what the request is and what parameters go with it, as specified in the USB specification, section 9.3. LUFA fills this information into a structure which has exactly the data fields described in that USB section. This structure is called — not surprisingly — `USB_ControlRequest`. All information about the request is therefore available to the user when the control request event is fired, ready to be processed.

This behaviour is described in the LUFA documentation in Modules→USB Core→USB Events →EVENT_USB_Device_ControlRequest().

Now if the user application has decided to process the data, but let LUFA do its own processing as well, there is not more to do than exit the event handler. However, if LUFA may *not* do any further processing, the user application needs a way to tell LUFA about that. This is done by calling `Endpoint_ClearSETUP()`. This function clears the RXSTPI flag, which has three effects:

- the setup packet is ACKed to the host,
- the endpoint bank is cleared,

Clearing the control endpoint bank is important, as it frees the endpoint for future USB packets.

- LUFA will not touch the request any more or try to process it.

So in the end, the user must know two things about the setup transaction: LUFA fires an event when a setup token and the associated setup packet have been received. The user can decide to process it and, independently, choose to acknowledge the packet.

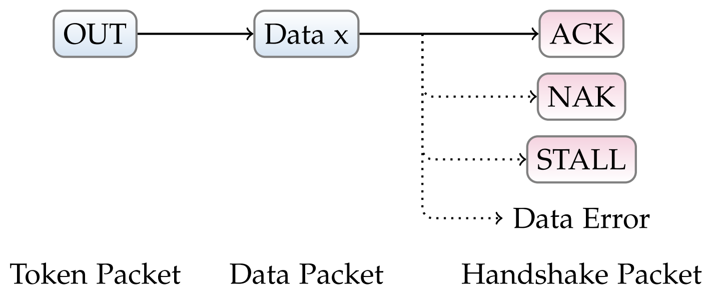
What happens if the control request contained invalid data? For any further USB traffic to happen, the control endpoint bank must be cleared, which also acknowledges the setup packet. The only way to signal to the host that there was an error is during the status transaction, which is placed at the end of the whole control transfer, after the optional data transaction. We will first go into detail about the data transaction, and then get to the status transaction.

Error management is not part of this tutorial. How it can be implemented depends on the application.

*The Data Transaction*

When the setup packet has been received, the direction of the data transaction is also known. It is specified in bit 7 of the `bmRequestType` field of the setup packet, which is of course also included in `USB_ControlRequest`. If the direction is from host to device, it starts with an OUT token:

The request type flags can be extracted with the macros described in LUFA→Modules→USB Core→Standard USB Requests.



The control endpoint size is limited to 8 bytes, so if the host wants to write more data than that, multiple data transactions are necessary. Each transaction then starts with an OUT token, followed by the data packet and a handshake packet. The device can, instead of ACKing the data, send a NAK or STALL, or let the packet time out.

The setup data contains exactly 8 bytes, which is why the endpoint size limitation was not of any relevance for the setup transaction.

LUFA provides a number of different ways to receive the data packets. The main difference is between the functions that can operate on any endpoint (including endpoint zero) and those which have been written specifically for the control endpoint. The latter take into account that the data transactions are followed by a status transaction. Those functions will be discussed when the status transaction has been explained.

`Endpoint_Read_xx(...)`, `Endpoint_Read_Stream_xx(...)`: see LUFA→USB Core→Endpoint Management.

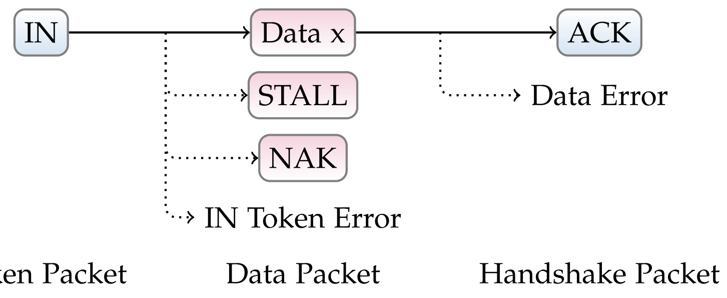
The "normal" endpoint functions can again be split into two groups: those which handle primitive data types and those which handle stream–like data. In either case, an ACK is sent to the host when a full packet has been received. The last packet, however, will not be "finished", *even if it was a full packet*. This has to be done by the user, who needs to send a final ACK. LUFA provides the function `Endpoint_ClearOUT()` for that.

`Endpoint_ClearOUT()`: see LUFA→USB Core→Endpoint Management→Endpoint Packet Management

The advantage of not sending the last ACK automatically is that the user application can direct the received data to several different destinations (scattering) by using multiple calls to (possibly different) receive functions. Only the end of the transmission must be indicated to the host. The user application knows how many bytes the host will send, as the host has to send *exactly* the number of bytes indicated in the `wLength` field of the setup packet.

See USB specification section 9.3.5 for more information about the `wLength` field.

If the `bmRequestType` of the setup packet field indicated an IN transaction, the device has to send data to the host:



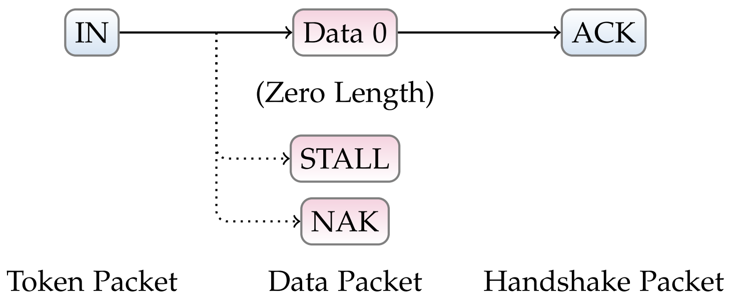
Again, it is possible that multiple data packets are needed, and the basic algorithm is the same as with host–to–device transactions. The relevant LUFA functions can be chained together, and will clear the IN endpoint bank after each full packet, but not after the last packet. The final packet has to be sent explicitly by calling `Endpoint_ClearIN()`.

`Endpoint_Write_xx(...)`, `Endpoint_Write_Stream_xx(...)`: see LUFA→USB Core→Endpoint Management.

`Endpoint_ClearIN()`: see LUFA→USB Core→Endpoint Management→Endpoint Packet Management.

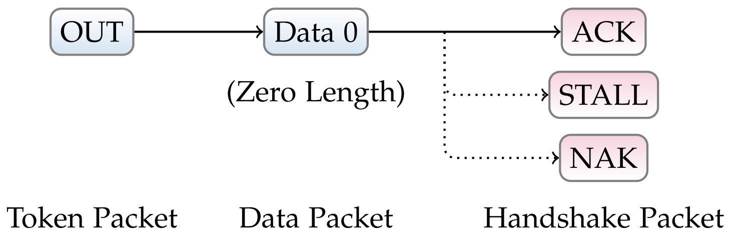
*The Status Transaction*

When the data has been transferred to or from the device, the control transfer is finished with a status transaction. When the host sent data to the device, the status transaction starts with an IN token:



Essentially, the device ACKs the whole data transaction by sending an empty data packet to the host during the status stage. This is done by simply clearing the IN token by calling `Endpoint_ClearIN()`. The host must then ACK the empty data packet, but nothing more has to be done by the user application.

When the host previously received data from the device, the status transaction starts with an OUT token:



This time the host ACKs the whole data transaction with an empty data packet, and the user application must ACK the empty data packet by calling `Endpoint_ClearOUT()`.

*LUFA functions for the Control Endpoint*

Many control transfers end up with a report being sent to the device or to the host. These might have been stored in a structure, an array, or whatever else contiguous memory region — they could be written to the endpoint with a single call to the various read or write functions and wouldn't require multiple calls. This is the situation where LUFA's control endpoint–specific functions are convenient.

These functions are used in the data stage of the control transfer, instead of the normal read or write functions. They send or receive full packets as needed, but — in contrast to the normal read or write functions — also clear the endpoint bank after the final packet. The application can directly proceed with the status stage by finally clearing the status transaction. Another convenience function is provided here: `Endpoint_ClearStatusStage()`, which internally calls `Endpoint_ClearIN()` or `Endpoint_ClearOUT()`, depending on the direction bit received in the setup packet.

`Endpoint_Write_Control_Stream_xx(...)`, `Endpoint_Read_Control_Stream_xx(...)`: see LUFA→USB Core→Endpoint Management.

`Endpoint_ClearStatusStage()`: see LUFA→USB Core→Endpoint Management.

In the next section we'll put together an interface for a servo controller, using different kinds of LUFA functions to manage control transfers. Both the normal functions and those specific to control endpoints will be used.

### *Example Code*

Every LUFA–based application must call `USB_Task()` periodically in order to handle USB. This function processes all USB–related events and calls the appropriate event handlers. One of them is `EVENT_USB_Device_ControlRequest()` (this was outlined above). Exactly this handler has to be filled with life now. We start by outlining the control requests we want to handle and how these are identified once they have been received.

An RC servo controller might have 4 basic functions:

*`void Servo_set(uint8_t channel, uint16_t value):`* This function sets one channel to the given value;

The allowed range for servo range is assumed to be 0 (full left) to 65 535 (full right).

*`void Servo_setAll(uint16_t* values):`* This function sets all servo channels to the values pointed at by `values`;

*`uint16_t Servo_get(uint8_t channel):`* This function returns the value of a given channel;

*`void Servo_getAll(uint16_t* dest):`* This function reads all servo channels and returns them in the array pointed at by `dest`.

These functions need an equivalent on the USB side of things, as USB doesn't know about these functions. This is done with a bunch of constants:

```
#define SERVO_CMD_SET 0
#define SERVO_CMD_SETALL 1
#define SERVO_CMD_GET 2
#define SERVO_CMD_GETALL 3
```

These constants will also be needed on the PC side, so it is a good idea to put them into a seperate header file!

These constants will indicate the desired operation in the `bRequest` field of the setup data. But there are more fields in the request structure to be analysed before the application code can decide which, if any, action is to be taken. A close look at the constants used above reveals that `SERVO_CMD_SET` is equal to `GET_STATUS`, which is a standard device request. The standard device requests are *standard* requests defined in the USB specification, and we are not allowed to override them. However, we are free to implement *vendor specific* or *class specific* requests. This is indicated in the `bmRequestType` field of the setup data along with the recipient of the request, which can be the device, one of the device's interfaces, an endpoint or even some unspecified recipient.

The standard device requests are described in the USB specification, section 9.4.

LUFA provides macros for extracting information about the request type and recipient. See LUFA→USB Core→Standard USB Requests.

For now, all commands are class specific and addressed to the device. The "set" requests are host–to–device, and the "get" requests are device–to–host. All these selection criteria can be put into our application's control request handler, which then chooses to act on the request, or not (effectively passing the request on to LUFA, which then tries to process it as a standard request):

This is by no means some kind of "best practice." It works, that's all.

```
void EVENT_USB_Device_ControlRequest()
{
  if(((USB_ControlRequest.bmRequestType & CONTROL_REQTYPE_TYPE)
      == REQTYPE_CLASS)
   && ((USB_ControlRequest.bmRequestType & CONTROL_REQTYPE_RECIPIENT)
      == REQREC_DEVICE))
  {
    if ((USB_ControlRequest.bmRequestType & CONTROL_REQTYPE_DIRECTION)
        == REQDIR_HOSTTODEVICE)
    {
      switch(USB_ControlRequest.bRequest)
      {
        case SERVO_CMD_SET:
          process_SERVO_CMD_SET();
          break;
        case SERVO_CMD_SETALL:
          process_SERVO_CMD_SETALL();
          break;
      }
    }
    else
    {
      switch(USB_ControlRequest.bRequest)
      {
        case SERVO_CMD_GET:
          process_SERVO_CMD_GET();
          break;
        case SERVO_CMD_GETALL:
          process_SERVO_CMD_GETALL();
          break;
      }
    }
  }
}
```

The control request handler first checks if the request is class specific and addressed to the device (that is, not to an interface or an endpoint). It then narrows down the request in switch statements, one for each direction. If no match is found, the request is not handled, but passed to LUFA instead. Again, this could be refined with error checking and reporting.

The first command (SERVO_CMD_SET) has only two parameters, a channel number and a value, which fit in the setup data structure's wIndex and wValue fields, respectively. The command is finally handled in its own function:

These fields are used in standard requests to select device descriptors, strings and such. They are used here in an analogous way; the parameters could also be passed as part of the data transaction.

```
void process_SERVO_CMD_SET()
{
  /* marks the command as "accepted" by the
  application, so that LUFA does not process it: */
  Endpoint_ClearSETUP();
  /* mark the whole request as successful: */
  Endpoint_ClearStatusStage();
  /* process command parameters: */
  Servo_set((USB_ControlRequest.wIndex & 0x07),
            USB_ControlRequest.wValue);
}
```

The index is masked with 0x07 to prevent memory corruption above an array with 8 elements.

In this function, it is assumed that the channel number in the wIndex field is valid. There is no error checking and handling. There is also no data transaction after the setup data has been transmitted, because all parameters fit into the setup data structure.

The next command is used to set all servo positions at once. The channel information is not necessary any more, but the overall amount of parameter data has increased to 16 bytes:

```
void process_SERVO_CMD_SETALL()
{
  uint16_t uiServoValues[8];
  Endpoint_ClearSETUP();
  /* read data from endpoint */
  Endpoint_Read_Stream_LE(uiServoValues, 16, 0);
  /* clear last data packet */
  Endpoint_ClearOUT();
  /* wait for the final IN token: */
  while (!(Endpoint_IsINReady()));
  /* and mark the whole request as successful: */
  Endpoint_ClearIN();
  /* process command parameters: */
  Servo_setAll(uiServoValues);
}
```

The thrid argument of Endpoint_Read_Stream_LE is used for monitoring the progress of longer, multi–packet transactions. We simply don't need it here. Zero is a safe value in this case.

Endpoint_IsINReady(): see LUFA→USB Core→Endpoint Management→Endpoint Packet Management.

This variant used the generic endpoint read function and not the function specific to control endpoints and also does not use the convenience function for clearing the status transaction. In this case Endpoint_ClearIN() is used: the host sends an IN token after all data packets have been sent to the device, which is in turn the request to the device to ACK the whole control transfer with an empty data packet. This request is ACKed with the final call to `Endpoint_ClearIN()`. Again, there is no error checking.

The above function can be rewritten using the provided convenience functions:

```
void process_SERVO_CMD_SETALL()
{
  uint16_t uiServoValues[8];
  Endpoint_ClearSETUP();
  /* read data from endpoint */
  Endpoint_Read_Control_Stream_LE(uiServoValues, 16);
  /* and mark the whole request as successful: */
  Endpoint_ClearStatusStage();
  /* process command parameters: */
  Servo_setAll(uiServoValues);
}
```

There are really no surprises in the "`get`" functions. For single channels:

```
void process_SERVO_CMD_GET()
{
  /* get value from servo driver */
  uint16_t uiServoValue = Servo_get(USB_ControlRequest.wIndex);
  Endpoint_ClearSETUP();
  /* write data to endpoint */
  Endpoint_Write_16_LE(uiServoValue);
  /* send packet */
  Endpoint_ClearIN();
  /* and mark the whole request as successful: */
  Endpoint_ClearStatusStage();
}
```

In this case, one of the functions for primitive data types has been used for writing data to the endpoint. We could have used the stream function instead, by replacing

```
Endpoint_Write_16_LE(uiServoValue);
```

with

```
Endpoint_Write_Stream_LE(&uiServoValue, 2, 0);
```

with no further changes to the code. The variant for eight channels works the same way, just the local variable for temporarily storing the values and the function for reading them are replaced:

```
void process_SERVO_CMD_GETALL()
{
  uint16_t uiServoValues[8];
  Endpoint_ClearSETUP();
  Servo_getAll(uiServoValues);
  Endpoint_Write_Stream_LE(uiServoValues, 16, 0);
  Endpoint_ClearIN();
  Endpoint_ClearStatusStage();
}
```

## *What's Left*

This was a tutorial about handling control transfers with LUFA. It was not about the general structure of a LUFA–powered application, and it didn't explain how to control RC servos with an AVR. The general structure of LUFA–powered applications can be seen in the numerous LUFA demos, like the mouse demo or its low–level equivalent. They also show how control transfers are handled in a bigger context.

There are many ways of controlling RC servos with an AVR. One of them can be picked out and adapted to the interface used in this tutorial. Some are easy to implement, some not; some eat up some special function of the AVR, some don't; some use external logic to simplify the code or improve timing. Most of them are worth knowing.

## *Revision History*

*1.1* Changes:

- Spelling, grammar and such,
- updated request type mask `CONTROL_REQTYPE_DIRECTION`, which had a different name in some previous LUFA release.

*1.0* Initial release