Application Note

BRT_AN_089

FT9XX USB Device Extended API

Version 1.0

Issue Date: 29-08-2023

This guide documents the use of the USB Device API for the FT9XX series devices from Bridgetek.

Bridgetek Pte Limited (BRTChip)
178 Paya Lebar Road, #07-03, Singapore 409030
Tel: +65 6547 4827
Web Site: http://www.brtchip.com

Table of Contents

1 Introduction

The FT9xx hardware library has a USB Device library API described in `ft900_usbd.h`. The source code is in the file `usbd.c`.

The FT9xx SDK before the v2.5.0 release used a polled API. From v2.5.0 onwards an interrupt-based API was implemented.

1.1 USB Device Polled API

It is designed to handle requests from the USB host by periodic polling of the USB device status from user code.

Requests from the USB host are received in one of multiple FIFO buffers (one for the control endpoint and one each for each of the numbered endpoints). Interrupts are generated for each packet received in this way and for each completed transmission event from the device to the host.

An interrupt service routine notes that there has been an interrupt and stores this information until the polling routine is activated. The polling routine will action the requests from the host and make calls to functions that provide device response to the host requests.

There are options for callback functions to notify user code of the successful (or otherwise) completion of these transfers. Callback functions can be setup for transfers, status changes or device events.

The callbacks are made from the periodic polling routine. Therefore, they will use the stack associated with the polling routine.

1.2 USB Device Interrupt API

This method uses an interrupt routine to action requests from the USB host and notify the user program of USB device status.

When host requests are received in one of the multiple FIFO buffers the interrupt handler is triggered and the handler actions the requests from the host during the interrupt handler.

Callbacks to user programs are also made by the interrupt handler. This means that the callback routines must complete as quickly as possible to release the MCU from interrupt handling.

1.2.1 Additional Functionality

There is an additional feature added to the interrupt compared to the polled API. A methodology of pipes and URBs (USB Request Block) is implemented to extend the flexibility of the device driver for reception and transmission of data.

The advantage of this is that data can be queued for reception or transmission asynchronously without programmatical control. This means that transfers can be scheduled and do not need any attention given to the USBD driver to control the operation.

1.3 Compatibility

The interrupt API is an evolution of the polled API to overcome certain issues. The interrupt API is generally backward compatible with the polled API.

Only the USBD_process polling function and the ep_cb parameter of the USBD_create_endpoint function is missing.

- The polling function is unnecessary when using the interrupt API as the interrupts do not need to be processed by user code.
- The callback is not used as it replaced by a similar method in the interrupt API.

The handling of SETUP packets is identical for both methods.

2 Comparison

The polled method and the interrupt method both provide capable solutions for implementing a USB device. The benefits and downsides of each method are:

2.1 USB Device Polled API

Benefits:

- Can debug USB transfers by stepping through USBD_process function calls as the MCU is not in interrupt mode.
- Callback functions run at normal priority allowing them to take longer to complete, meaning that they can perform more tasks and possibly access other interfaces for data.
- USBD_process function runs with the normal stack of the calling function. This can be useful for RTOS operation as the call can be made from a dedicated task or thread which has a suitable stack for the task.
- The interrupt handler is small and very fast. This may help other processes which rely on interrupt handling to process other tasks at the same time.
- Asynchronous operations can be performed, one at a time, by using an asynchronous method which allows for non-blocking reading and writing.

Disadvantages:

- The USBD_process must be called regularly to process all USB device state changes, requests, and transfers. If the calls are not made often enough, then the performance of the device will suffer.
- Reading and writing are normally blocking until the transfer completes. The asynchronous methods still require regular calls to USBD_process to make the transfers work.
- Asynchronous transfers cannot be queued.

2.2 USB Device Interrupt API

Benefits:

- No action is required in the user code to manage USB device state changes, requests and transfers.
- Response and transfer speed is increased since most actions are performed at interrupt level.
- Multiple transfers can be queued at a time to occur rapidly in order.

Disadvantages:

- The interrupt handler adds to the stack of the interrupted task. It might not be able to process as much data as required.
- All callback functions and handlers in user code are at interrupt level so they must complete as quickly as possible. There is no chance of requesting data from another interface while at interrupt level.

3 Choosing a USB API

It is possible to revert to the polling API by replacing the file usbd.c with one from a previous version in the hardware library.

Unless there are compelling reasons to use the polled API then the interrupt API should be chosen. It provides many benefits and simplifies receiving and transmitting data once it is setup correctly. It should also be able to send and receive data at higher bandwidths than the polled API.

4 Using the Interrupt API

This section discusses the implementation of the interrupt API. It is simply a set of arbitrarily allocated "pipes" which are assigned to a hardware endpoint. A memory structure representing a transfer, called a URB (USB Request Block), is associated with one of these pipes to trigger an operation of the USB device.

4.1 Pipes

The principle of the interrupt API is that there is a single "pipe" for each endpoint. These pipes are represented by a structure struct USBDX_pipe. The structure must be stored persistently (global or static local storage).

Since each pipe is associated exclusively with one endpoint, a convenient way of selecting a pipe for an endpoint is to use the unique endpoint number as a reference to the pipe.

The following is a function that can store the pipe structures and return a pointer to a single pipe for a specific endpoint.

```
enum {
	CDC_EP_NOTIFICATION = 1,
	CDC_EP_DATA_OUT,
	CDC_EP_DATA_IN,
	CDC_EP_MAX
};

static struct USBDX_pipe *get_pipe(uint8_t endpoint)
{
	static struct USBDX_pipe pipes[CDC_EP_MAX];
	struct USBDX_pipe *ret_val = NULL;

	switch (endpoint) {
		case CDC_EP_NOTIFICATION:
			ret_val = &pipes[0];
			break;
		case CDC_EP_DATA_IN:
			ret_val = &pipes[1];
			break;
		case CDC_EP_DATA_OUT:
			ret_val = &pipes[2];
			break;
		default:
			break;

	}
	assert(ret_val!=NULL);
	return ret_val;
}
```

Function 1 – Example of a function to store and find pipe structures

The example allows 3 endpoints, one for "notification", one in and out endpoint for "data". The 3 pipes can be found from only the endpoint number from within the code by calling this function.

4.2 Interrupts

An interrupt from the API is received within the function `USBDX_pipe_isr` which is defined with a weak function linkage in the library code.

This means that to receive interrupts for a pipe (endpoint) this function must be defined in the application.

The function receives a bitmap indicating all endpoints which have an interrupt pending. The bitmap is 16 bits wide and has a set bit in position *n* for endpoint number *n*.

To clarify, 0000,0000,0000,0010b is endpoint 1 (`CDC_EP_NOTIFICATION` in our example), 0000,0000,0000,1000b is endpoint 3

```
void USBDX_pipe_isr(uint16_t pipe_bitfields)
{
        if (pipe_bitfields & BIT(CDC_EP_NOTIFICATION))
                USBDX_pipe_process(get_pipe(CDC_EP_NOTIFICATION));
        if (pipe_bitfields & BIT(CDC_EP_DATA_IN))
                USBDX_pipe_process(get_pipe(CDC_EP_DATA_IN));
        if (pipe_bitfields & BIT(CDC_EP_DATA_OUT))
                USBDX_pipe_process(get_pipe(CDC_EP_DATA_OUT));
}
```

Function 2 – Example of a function to receive pipe and interrupts

The main feature of the interrupt handler is to pass the pipe structure allocated to the `USBDX_pipe_process` function. From there the pipe contents can be processed before handing data back to the application.

4.3 URBs

The URB represents an IN or an OUT transfer on the USB. There can be multiple URBs for each pipe (endpoint) always arranged as an array of URBs.

Every URB must include a data buffer which is of the maximum size of the endpoint which the URB will be supporting. There will be exactly one data buffer for each URB. If the data buffer is smaller than the endpoint maximum size then buffer overruns will occur when packets are received from the host.

If there are multiple URBs then the data buffers for each are arranged as an array of smaller buffers. Data for each URB will be read or written in the smaller buffer associated with the URB.

The URB is passed as a pointer to the static storage to the `USBDX_pipe_init` function with a pointer to the data buffer. A pointer to pipe structure is passed which is then associated with the URB.

The following is a typical initialization of 3 endpoints and 3 sets of URBs with data buffers.

```
#define ACM_URB_INT_COUNT  2
#define ACM_URB_IN_COUNT   4
#define ACM_URB_OUT_COUNT  4
#define CDC_NOTIFICATION_USBD_EP_SIZE 8
#define CDC_DATA_EP_SIZE_HS 512

USBD_create_endpoint(CDC_EP_NOTIFICATION, USBD_EP_INT, USBD_DIR_IN,
                     CDC_NOTIFICATION_USBD_EP_SIZE, USBD_DB_OFF, NULL /*ep_cb*/);
USBD_create_endpoint(CDC_EP_DATA_OUT, USBD_EP_BULK, USBD_DIR_OUT,
                     CDC_DATA_USBD_EP_SIZE_HS, USBD_DB_ON, NULL /*ep_cb*/);
USBD_create_endpoint(CDC_EP_DATA_IN, USBD_EP_BULK, USBD_DIR_IN,
                     CDC_DATA_USBD_EP_SIZE_HS, USBD_DB_ON, NULL /*ep_cb*/);

static uint8_t acm_int_buf[CDC_NOTIFICATION_EP_SIZE * ACM_URB_INT_COUNT];
static uint8_t acm_in_buf[CDC_DATA_EP_SIZE_HS * ACM_URB_IN_COUNT];
static uint8_t acm_out_buf[CDC_DATA_EP_SIZE_HS * ACM_URB_OUT_COUNT];

static struct USBDX_urb acm_int[ACM_URB_INT_COUNT];
static struct USBDX_urb acm_in[ACM_URB_IN_COUNT];
static struct USBDX_urb acm_out[ACM_URB_OUT_COUNT];

struct USBDX_pipe *pp;

pp = get_pipe(CDC_EP_NOTIFICATION);
USBDX_pipe_init(pp, CDC_EP_NOTIFICATION, CDC_EP_NOTIFICATION | 0x80,
                     acm_int, acm_int_buf, ACM_URB_INT_COUNT);

pp = get_pipe(CDC_EP_DATA_IN);
USBDX_pipe_init(pp, CDC_EP_DATA_IN, CDC_EP_DATA_IN | 0x80,
                     acm_in, acm_in_buf, ACM_URB_IN_COUNT);

pp = get_pipe(CDC_EP_DATA_OUT);
USBDX_pipe_init(pp, CDC_EP_DATA_OUT, CDC_EP_DATA_OUT,
                     acm_out, acm_out_buf, ACM_URB_OUT_COUNT);}
```

Function 3 – Example of setting up endpoints and URBs

There are 4 data buffers of size 512 bytes associated with the IN and OUT data endpoints but only 2 buffers of size 8 for the notification endpoint. The `acm_int_buf` is therefore 16 bytes and the `acm_in_buf` and `acm_out_buf` are both 2 kB.

4.3.1 Ownership

Each URB represents a transfer of the USB. While the URB is not being actively used it is termed to be owned by the application. Once it has been submitted to the driver, until it is completed then it is owned by usbdx. The `usbdx_urb_owned_by_app` function is used to tell the ownership of the transfer.

The ownership of the URB may change during an interrupt. By default, IN endpoints are normally owned by the application, OUT endpoints are owned by usbdx.

The reason for this is explained below:

- OUT endpoints receive data from the host asynchronously, the URB is owned by usbdx until the point where valid data has been received, it is then given to the application to process the data. Once the application has finished with both the URB and the data buffer then it can return it to usbdx for receiving later packets.
- IN endpoints are not active until the application has data which it wants to send to the host. Therefore, the application owns the URB until it has data to send. It will then transfer ownership to usbdx until the data has been sent to the host.

4.3.2 Sending data (IN endpoints)

The function `usbdx_get_app_urb` is used to find a URB that can be used for an IN transfer. It must be checked that it is owned by the application first with the `usbdx_urb_owned_by_app` function.

Once a free URB has been successfully found then the data to send can be written into the data buffer pointed to by the URB.

The pointer `urb->ptr` can be used as the start of the data buffer.

Once the data is copied then the `USBDX_submit_urb` function is used to pass control of the URB to the usbdx and commence transmission to the host.

The pipe will automatically pause once data has been sent.

4.3.3 Receiving data (OUT endpoints)

Once data has been received in a URB for an OUT endpoint the usbdx expects the application to process the data and reuse the URB.

To do this, the application registers a callback function for a pipe (endpoint). This will be called from the interrupt service routine when data is ready for that pipe. The callback must be registered like this below -

```
        USBDX_register_on_ready(pp, acm_out_on_data_ready);
```

Function 4 – Example of registering a data processing callback

The function registers the callback with a single pipe, however multiple pipes can use the same callback. A pointer to the pipe to process is passed to the callback function.

The callback function is defined as follows.

```
/* return true if any URB is submitted in callback function */
typedef bool (*USBDX_callback)(struct USBDX_pipe *pp);
```

Function 5 – Definition of data processing callback

The same function can also be used for an underrun condition. It is not necessary to handle underruns. The callback function will be called once data has been received from the host. Multiple URBs may be used to hold the data received:

```
bool acm_out_on_data_ready(struct USBDX_pipe *pp)
{
        do {
                USBDX_urb *urb = usbdx_get_app_urb(pp);
                if (!usbdx_urb_owned_by_app(urb))
                        break;

                uint16_t urb_len = usbdx_urb_get_app_to_process(urb);
                process_data(urb->ptr, urb_len);
                USBDX_submit_urb(pp, urb);
                urb = usbdx_get_app_urb(pp);
        } while (1);
        return true;
}
```

Function 6 – Example of data processing callback

The `process_data` function (not shown) copies the data out of the URB data buffer for use by the application and can signal the application through a flag or other notification method. The callback function is called at interrupt level so care must be taken if setting global variables from this state.
To receive data from the host, the `USBDX_pipe_init` will start reception into the available URBs.
If, the data has been processed and the application wishes to resubmit the URB immediately for receiving subsequent data then it may perform a `USBDX_submit_urb` request within the callback. It can even submit URBs to different pipes. If it does this then it must return true. If it does not submit a URB then it must return false and the endpoint will be temporarily halted until another `USBDX_submit_urb` function call.

5 Initialisation Process

The USBD_init function and the USBD_ctx structure are initialised as normal.

```
        USBD_ctx usb_ctx;

        memset(&usb_ctx, 0, sizeof(usb_ctx));

        usb_ctx.standard_req_cb = NULL;
        usb_ctx.get_descriptor_cb = standard_req_get_descriptor;
        usb_ctx.class_req_cb = class_req_cb;
        usb_ctx.vendor_req_cb = vendor_req_cb;
        usb_ctx.suspend_cb = suspend_cb;
        usb_ctx.resume_cb = resume_cb;
        usb_ctx.reset_cb = reset_cb;
        usb_ctx.lpm_cb = NULL;
        usb_ctx.speed = USBD_SPEED_HIGH;

        // Initialise the USB device with a control endpoint size
        // of 8 to 64 bytes. This must match the size for bMaxPacketSize0
        // defined in the device descriptor.
        usb_ctx.ep0_size = USB_CONTROL_EP_SIZE;

        USBD_initialise(&usb_ctx);
```

Function 7 – Example of registering a data processing callback

Next, the application can (but does not have to) wait for a connection to the host. If the device is bus-powered then it will mostly likely be able to wait for connection by the host.

```
        while (!USBD_is_connected())
        {
        }
        USBD_attach();
        USBD_connect();
```

Function 8 – Example of registering a data processing callback

At this point the endpoints, pipes and URBs can be created and initialised as in Figure xxx above.

It is important to wait until the USB device is attached to the host so that the bus speed can be determined. For some devices the device descriptor, configuration descriptor and possibly other descriptors are affected by the bus speed. Waiting until this information is found is necessary if, for instance, the maximum packet size for endpoint zero is greater than 8 bytes for a High-Speed device and 8 bytes for a Full-Speed device.

6 Sample Code

Checklist:

- static storage for every pipe structure.

- method of correctly selecting the pipe structure from an endpoint number and bitmap.

- static storage for URB structures associated with a pipe.

- static storage for URB data structures which are adequate size for endpoint data transfers.

- URB structures correctly associated with a single pipe structure.

- Correct use of `USBD_create_endpoint` before `USBDX_pipe_init` function call.

- function `USBDX_pipe_isr` exists in user code and passes correct pipe structure to `USBDX_pipe_process`.

7 Contact Information

Refer to https://brtchip.com/contact-us/ for contact information.

Distributor and Sales Representatives

Please visit the Sales Network page for the contact details of our distributor(s) and sales representative(s) in your country.

Appendix A – References

Document References

https://brtchip.com/ft9xx-toolchain/

AN_360 FT9XX Example Applications

TN_160 Eclipse Projects

Acronyms and Abbreviations

Terms	Description
CMD	Command-line interface
DLL	Dynamic-link Library
DLOG	Data Log (Project)
GAS	GNU Assembler
GCC	GNU Compiler Collection
GDB	GNU Project Debugger
GNU	GNU (Gnu’s Not Unix) Operating System
GUI	Graphical User Interface
IDE	Integrated Development Environment
JDK	Java Development Kit
JRE	Java Runtime Environment
MCU	Microcontroller Unit
PATH	PATH Environment Variable
TCP	Transmission Control Protocol

Appendix B – List of Tables & Figures

List of Tables

NA

List of Figures

NA

List of Functions

Appendix C – Revision History

Document Title:	BRT_AN_089 FT9XX USB Device Extended API
Document Reference No.:	BRT_000421
Clearance No.:	BRT#199
Product Page:	https://brtchip.com/ft9xx-toolchain/
Document Feedback:	Send Feedback

Revision	Changes	Date
1.0	Initial release	29-08-2023