Linux Driver Development for Embedded Processors

ALBERTO LIBERAL DE LOS RÍOS

13
Linux USB Device Drivers

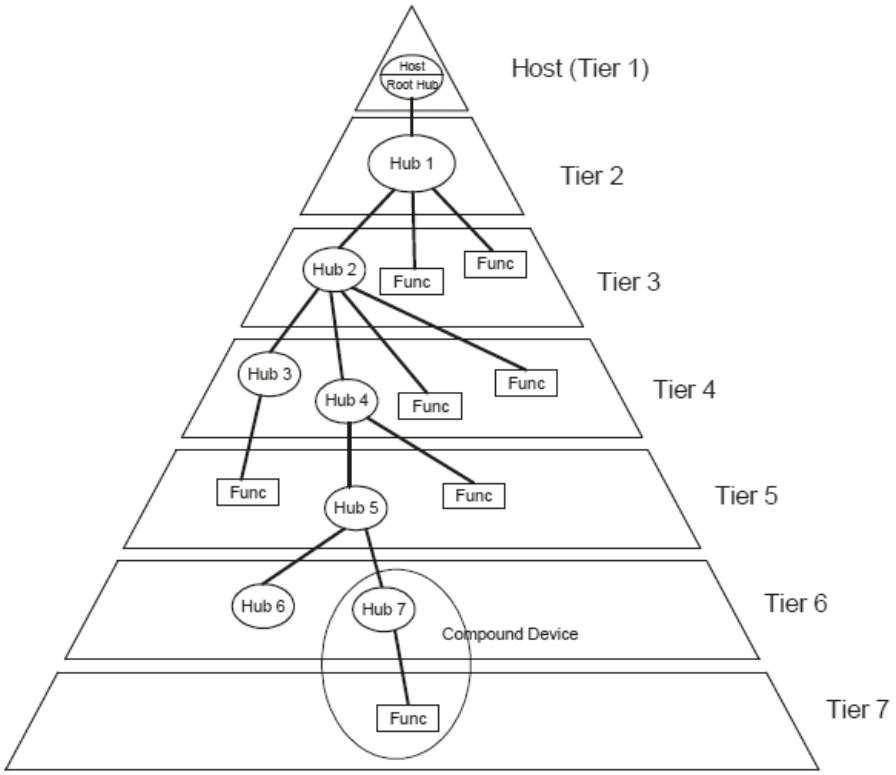
USB (abbreviation of **Universal Serial Bus**) was designed as a low cost, serial interface solution with bus power provided from the USB host to support a wide range of peripheral devices. The original bus speeds for USB were low speed at 1.5 Mbps, followed by full speed at 12 Mbps, and then high speed at 480 Mbps. With the advent of the USB 3.0 specification, the super speed was defined at 4.8 Gbps. Maximum data throughput, i.e. the line rate minus overhead is approximately 384 Kbps, 9.728 Mbps, and 425.984 Mbps for low, full and high speed respectively. Note that this is the maximum data throughput and it can be adversely affected by a variety of factors, including software processing, other USB bandwidth utilization on the same bus, etc.

One of the biggest advantages of USB is that it supports dynamic attachment and removal, which is a type of interface referred to as "plug and play". Following attachment of a USB peripheral device, the host and device communicate to automatically advance the externally visible device state from the attached state through powered, default, addressed and finally to the configured states. Additionally, all devices must conform to the suspend state in which a very low bus power consumption specification must be met. Power conservation in the suspended state is another USB benefit.

Throughout this chapter, we will focus on the USB 2.0 specification, which includes low, full and high speed device specifications. Compliance to USB 2.0 specification for peripheral devices does not necessarily indicate that the device is a high speed device, however a hub advertised as USB 2.0 compliant, must be high speed capable. A USB 2.0 device can be High Speed, Full Speed, or Low Speed.

USB 2.0 Bus Topology

USB devices fall into the category of hubs - which provide additional downstream attachment points, or functions - which provide a capability to the system. USB physical interconnection is a tiered star topology (see next Figure). Starting with the host and "root hub" at tier 1, up to seven tiers with a maximum of 127 devices can be supported. Tier 2 through 6 may have one or more hub devices in order to support communication to the next tier. A compound device (one which has both a hub and peripheral device functions) may not occupy tier 7.



The physical USB interconnection is accomplished for all USB 2.0 (up to high speed) devices via a simple 4-wire interface with bi-directional differential data (D+ and D-), power (VBUS) and ground. The VBUS power is nominally +5V. An "A-type" connector and mating plug are used for all host ports as well as downstream facing hub ports. A "B-type" connector and mating plug are used for all peripheral devices as well as the upstream facing port of a hub. Cable connections between host, hubs and devices can each be a maximum of 5 meters or ~16 feet. With the maximum of 7 tiers, cabling connections can be up to 30 meters or ~ 98 feet total.

USB Bus Enumeration and Device Layout

USB is a Host-controlled polled bus where all transactions are initiated by the USB host. Nothing on the bus happens without the host first initiating it; the USB devices cannot initiate a transaction, it is the host which polls each device, requesting data or sending data. All attached and removed USB devices are identified by a process termed "bus enumeration".

An attached device is recognized by the host and its speed (low, full or high) is identified via a signaling mechanism using the D+/D- USB data pair. When a new USB device is connected to the bus through a hub the device enumeration process starts. Each hub provides an IN endpoint, which is used to inform the host about newly attached devices. The host continually polls on this endpoint to receive device attachment and removal events from the hub. Once a new device was attached and the hub notified the host about this event, the USB bus driver of the host enables the attached device and starts requesting information from the device. This is done with standard USB requests which are sent through the default control pipe to endpoint zero of the device.

Information is requested in terms of **descriptors**. USB descriptors are data structures that are provided by devices to describe all of their attributes. This includes e.g. the product/vendor ID, any device class affiliation, and strings describing the product and vendor. Additionally information about all available endpoints is provided. After the host read all the necessary information from the device it tries to find a matching device driver. The details of this process are dependant on the used operating system. After the first descriptors were read from the attached USB device, the host uses the vendor and product ID from the device descriptor to find a matching device driver.

The attached device will initially utilize the default USB address of 0. Additionally, all USB devices are comprised of a number of independent endpoints which provide a terminus for communication flow between the host and device.

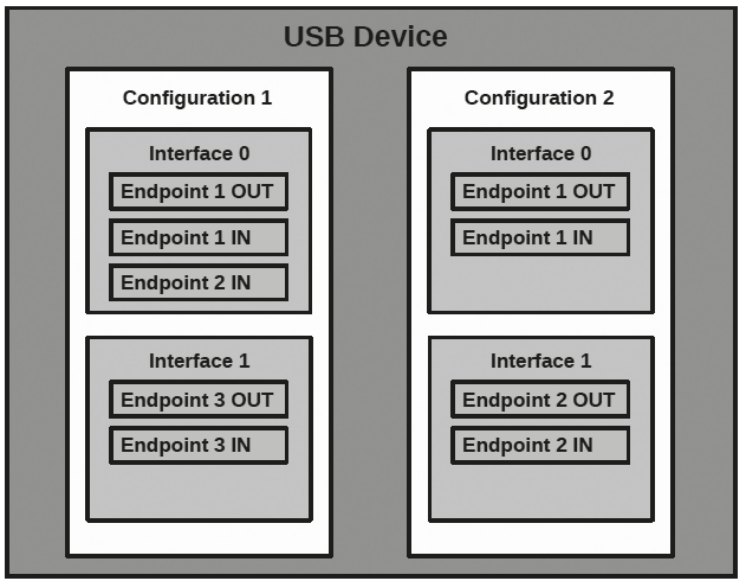
Endpoints can be categorized into **control** and **data** endpoints. Every USB device must provide at least one control endpoint at address 0 called the default endpoint or Endpoint0. This endpoint is bidirectional, that is, the host can send data to the endpoint and receive data from it within one transfer. The purpose of a control transfer is to enable the host to obtain device information, configure the device, or perform control operations that are unique to the device.

The endpoint is a buffer that typically consists of a block of memory or registers which stores received data or contain data which is ready to transmit. Each endpoint is assigned a unique endpoint number determined at design time, however all devices must support the default control endpoint (ep0) which is assigned number 0 and may transfer data in both directions. All other endpoints may transfer data in one direction (always from the host perspective), labeled "Out", i.e. data from the host, or "In", i.e. data to the host. The endpoint number is a 4-bit integer associated with an endpoint (0-15); the same endpoint number is used to describe two endpoints, for instance EP 1 IN and EP 1 OUT. An endpoint address is the combination of an endpoint number and an endpoint direction, for example: EP 1 IN, EP 1 OUT, EP 3 IN. The endpoint addresses are encoded with the direction and number in a

single byte, where the direction is the MSB (1=IN, 0=OUT) and number is the lower four bits. For example:

- EP 1 IN = 0x81
- EP 1 OUT = 0x01
- EP 3 IN = 0x83
- EP 3 OUT = 0x03

An USB configuration defines the capabilities and features of a device, mainly its power capabilities and interfaces. The device can have multiple configurations, but only one is active at a time. A configuration can have one or more USB interfaces that define the functionality of the device. Typically, there is a one-to-one correlation between a function and an interface. However, certain devices expose multiple interfaces related to one function. For example, a USB device that comprises a keyboard with a built-in speaker will offer an interface for playing audio and an interface for key presses. In addition, the interface contains alternate settings that define the bandwidth requirements of the function associated with the interface. Each interface contains one or more endpoints, which are used to transfer data to and from the device. To sum up, a group of endpoints form an interface, and a set of interfaces constitutes a configuration in the device. The following image shows a multiple-interfaces USB device:



After a matching device driver was found and loaded, it's the task of the device driver to select one of the provided device configurations, one or more interfaces within that configuration, and an alternate setting for each interface. Most USB devices don't provide multiple interfaces or multiple alternate settings. The device driver selects one of the configurations based on its own capabilities and the available bandwidth on the bus and activates this configuration on the attached device. At this point, all interfaces and their endpoints of the selected configuration are set up and the device is ready for use.

Communication from the host to each device endpoint uses a communication "pipe" which is established during enumeration. The pipe is a logical association between the host and the device. Pipe is purely a software term. A pipe talks to an endpoint on a device, and that endpoint has an address. The other end of a pipe is always the host controller. A pipe for an endpoint is opened when the device is configured either by selecting a configuration and an interface's alternate setting. Therefore they become targets for I/O operations. A pipe has all the properties of an endpoint, but it is active and be used to communicate with the host. The combination of the device address, endpoint number and direction allows the host to uniquely reference each endpoint.

USB Data Transfers

Once the enumeration is complete, the host and device are free to carry out communications via data transfers from the host to the device or viceversa. Both directions of transfers are initiated by the host. Four different types of transfers are defined. These types are:

- **Control Transfers**: Used to configure a device at attach time and can be used for other device-specific purposes, for example device specific register read/write access as well as control of other pipes on the device. Control transfers consist of up to three distinct stages, a setup stage containing a request, a data stage if necessary to or from the host, and a status stage indicating the success of the transfer. USB has a number of standardized transactions that are implemented by using control transfers. For example, the "Set Address" and "Get Descriptor" transactions are always utilized in the device enumeration procedure described above. The "Set Configuration" request is another standard transaction which is also used during device enumeration.
- **Bulk Data Transfers**: Capable of transfering relatively large quantities of data or bursty data. Bulk transfers do not have guaranteed timing, but can provide the fastest data transfer rates if the USB bus is not occupied by other activity.
- **Interrupt Data Transfers**: Used for timely but reliable delivery of data, for example, characters or coordinates with human-perceptible echo or feedback response characteristics. Interrupt transfers have a guaranteed maximum latency, i.e. time between transaction attempts. USB mice and keyboards typically use interrupt data transfers.

- **Isochronous Data Transfers**: Occupy a prenegotiated amount of USB bandwidth with a prenegotiated delivery latency. Isochronous transfers have guaranteed timing but do not have error correction capability. Isochronous data must be delivered at the rate received to maintain its timing and additionally may be sensitive to delivery delays. A typical use for isochronous transfers would be for streaming audio or video.

USB Device Classes

The USB specification and supplemental documents define a number of device classes that categorize USB devices according to capability and interface requirements. When a host retrieves device information, class classification helps the host determine how to communicate with the USB device. The hub is a specially designated class of devices that has additional requirements in the USB specification. Other examples of classes of peripheral devices are human interface, also known as HID, printer, imaging, mass storage and communications. The USB UART devices usually fall into the communications device class (CDC) of USB devices.

Human Interface Device Class

The HID class devices usually interface with humans in some capacity. HID-class devices include mice, keyboards, printers, etc. However, the HID specification merely defines basic requirements for devices and the protocol for data transfer, and devices do not necessarily depend on any direct human interaction. HID devices must meet a few general requirements that are imposed to keep the HID interface standardized and efficient.

- All HID devices must have a control endpoint (Endpoint 0) and an interrupt IN endpoint. Many devices also use an interrupt OUT endpoint. In most cases, HID devices are not allowed to have more than one OUT and one IN endpoint.
- All data transferred must be formatted as reports whose structure is defined in the report descriptor.
- HID devices must respond to standard HID requests in addition to all standard USB requests.

Before the HID device can enter its normal operating mode and transfer data with the host, the device must properly enumerate. The enumeration process consists of a number of calls made by the host for descriptors stored in the device that describe the device's capabilities. The device must respond with descriptors that follow a standard format. Descriptors contain all basic information about a device. The USB specification defines some of the descriptors retrieved, and the HID specification defines other required descriptors. The following section discusses the descriptor structures a host expects to receive.

USB Descriptors

The host software obtains descriptors from an attached device by sending various standard control requests to the default endpoint during enumeration, inmediately upon a device being attached. Those requests specify the type of descriptor to retrieve. In response to such requests, the device sends descriptors that include information about the device, its configurations, interfaces and the related endpoints. Device descriptors contain information about the whole device.

Every USB device exposes a **device descriptor** that indicates the device's class information, vendor and product identifiers, and number of configurations. Each configuration exposes it's **configuration descriptor** that indicates number of interfaces and power characteristics. Each interface exposes an **interface descriptor** for each of its alternate settings that contains information about the class and the number of endpoints. Each endpoint within each interface exposes **endpoint descriptors** that indicate the endpoint type and the maximum packet size.

Descriptors begin with a byte describing the descriptor length in bytes. This length equals the total number of bytes in the descriptor including the byte storing the length. The next byte indicates the descriptor type, which allows the host to correctly interpret the rest of the bytes contained in the descriptor. The content and values of the rest of the bytes are specific to the type of descriptor being transmitted. Descriptor structure must follow specifications exactly; the host will ignore received descriptors containing errors in size or value, potentially causing enumeration to fail and prohibiting further communication between the device and the host.

USB Device Descriptors

Every Universal Serial Bus (USB) device must be able to provide a single device descriptor that contains relevant information about the device. For example, the **idVendor** and **idProduct** fields specify vendor and product identifiers, respectively. The **bcdUSB** field indicates the version of the USB specification to which the device conforms. For example, 0x0200 indicates that the device is designed as per the USB 2.0 specification. The **bcdDevice** value indicates the device defined revision number. The device descriptor also indicates the total number of configurations that the device supports. You can see below an example of a structure that contains all the device descriptor fields:

```
typedef struct __attribute__ ((packed))
{
    uint8_t bLength;               // Length of this descriptor.
    uint8_t bDescriptorType;       // DEVICE descriptor type
(USB_DESCRIPTOR_DEVICE).
    uint16_t bcdUSB;               // USB Spec Release Number (BCD).
    uint8_t bDeviceClass;          // Class code (assigned by the USB-IF).
0xFF-Vendor specific.
```

```
    uint8_t bDeviceSubClass;        // Subclass code (assigned by the USB-IF).
    uint8_t bDeviceProtocol;        // Protocol code (assigned by the USB-IF).
0xFF-Vendor specific.
    uint8_t bMaxPacketSize0;        // Maximum packet size for endpoint 0.
    uint16_t idVendor;              // Vendor ID (assigned by the USB-IF).
    uint16_t idProduct;             // Product ID (assigned by the manufacturer).
    uint16_t bcdDevice;             // Device release number (BCD).
    uint8_t iManufacturer;          // Index of String Descriptor describing the
manufacturer.
    uint8_t iProduct;               // Index of String Descriptor describing the
product.
    uint8_t iSerialNumber;          // Index of String Descriptor with the device's
serial number.
    uint8_t bNumConfigurations;     // Number of possible configurations.

} USB_DEVICE_DESCRIPTOR
```

The first item **blength** describes the descriptor length and should be common to all USB device descriptors.

The item **bDescriptorType** is the constant one-byte designator for device descriptors and should be common to all device descriptors.

The BCD-encoded two-byte **bcdUSB** item tells the system which USB specification release guidelines the device follows. This number might need to be altered in devices that take advantage of additions or changes included in future revisions of the USB specification, as the host will use this item to help determine what driver to load for the device.

If the USB device class is to be defined inside the device descriptor, this item **bDeviceClass** would contain a constant defined in the USB specification. Device classes defined in other descriptors should set the device class item in the device descriptor to 0x00.

If the device class item discussed above is set to 0x00, then the device **bDeviceSubClass** item should also be set to 0x00. This item can tell the host information about the device's subclass setting.

The item **bDeviceProtocol** can tell the host whether the device supports high-speed transfers. If the above two items are set to 0x00, this one should also be set to 0x00.

The item **bMaxPacketSize0** tells the host the maximum number of bytes that can be contained inside a single control endpoint transfer. For low-speed devices, this byte must be set to 8, while full-speed devices can have maximum endpoint 0 packet sizes of 8, 16, 32, or 64.

The two-byte item **idVendor** identifies the vendor ID for the device. Vendor IDs can be acquired through the USB.org website. Host applications will search attached USB devices' vendor IDs to find a particular device needed for an application.

Like the vendor ID, the two-byte item **idProduct** uniquely identifies the attached USB device. Product IDs can be acquired through the USB.org web site.

The item **bcdDevice** is used along with the vendor ID and the Product ID to uniquely identify each USB device.

The next three one-byte items tell the host which string array index to use when retrieving UNICODE strings describing attached devices that are displayed by the system on-screen. This string describes the manufacturer of the attached device. An **iManufacturer** string index value of 0x00 indicates to the host that the device does not have a value for this string stored in memory.

The index **iProduct** will be used when the host wants to retrieve the string that describes the attached product. For example the string could read "USB Keyboard".

The string pointed to by the index **iSerialNumber** can contain the UNICODE text for the product's serial number.

This item **bNumConfigurations** tells the host how many configurations the device supports. A configuration is the definition of the device's functional capabilities, including endpoint configuration. All devices must contain at least one configuration, but more than one can be supported.

USB Configuration Descriptor

The USB device can have several different configurations although the majority of devices are simple and only have one. The configuration descriptor specifies how the device is powered, what the maximum power consumption is, the number of interfaces it has. Therefore it is possible to have two configurations, one for when the device is bus powered and another when it is mains powered. As this is a "header" to the interface descriptors, it is also feasible to have one configuration using a different transfer mode to that of another configuration. You can see below an example of a structure that contains all the configuration descriptor fields:

```
typedef struct __attribute__ ((packed))
{
    uint8_t bLength;             // Size of Descriptor in Bytes
    uint8_t bDescriptorType;     // Configuration Descriptor (0x02)
    uint16_t wTotalLength;       // Total length in bytes of data returned
    uint8_t bNumInterfaces;      // Number of Interfaces
    uint8_t bConfigurationValue; // Value to use as an argument to select this
configuration
    uint8_t iConfiguration;      // Index of String Descriptor describing this
configuration
    uint8_t bmAttributes;        // power parameters for the configuration
    uint8_t bMaxPower;           // Maximum Power Consumption in 2mA units

} USB_CONFIGURATION_DESCRIPTOR;
```

The item **blenght** defines the length of the configuration descriptor. This is a standard length.

The item **bDescriptorTye** is the constant one-byte 0x02 designator for configuration descriptors.

The two-byte **wTotalLength** item defines the length of this descriptor and all of the other descriptors associated with this configuration. For example, the length could be calculated by adding the length of the configuration descriptor, the interface descriptor, the HID class descriptor, and two endpoint descriptors associated with this interface. This two-byte item follows a "little endian" data format. The item defines the length of this descriptor and all of the other descriptors associated with this configuration.

The **bNumInterfaces** item defines the number of interface settings contained in this configuration.

The **bConfigurationValue** item is used by the SetConfiguration request to select this configuration.

The **iConfiguration** item is a index to a string descriptor describing the configuration in human readable form.

The **bmAttributes** item tells the host whether the device supports USB features such as remote wake-up. Item bits are set or cleared to describe these conditions. Check the USB specification for a detailed discussion on this item.

The **bMaxPower** item tells the host how much current the device will require to function properly at this configuration.

USB Interface Descriptor

An interface descriptor contains information about an alternate setting of an USB interface. Interface descriptors have a **bInterfaceNumber** field specifying the interface number and a **bAlternateSetting** which allows an interface to change settings on the fly. For example, we could have a device with two interfaces, interface one and interface two. Interface one has **bInterfaceNumber** set to zero indicating it is the first interface descriptor and a **bAlternativeSetting** of zero. Interface two would have a **bInterfaceNumber** set to one indicating it is the second interface and a **bAlternativeSetting** of zero (default). We could then throw in another descriptor, also with a **bInterfaceNumber** set to one indicating it is the second interface, but this time setting the **bAlternativeSetting** to one, indicating this interface descriptor can be an alternative setting to that of the other interface descriptor two.

The **bNumEndpoints** item indicates the number of endpoints used by the interface. This value should exclude endpoint zero and is used to indicate the number of endpoint descriptors to follow.

The **bInterfaceClass**, **bInterfaceSubClass** and **bInterfaceProtocol** items can be used to specify supported classes (e.g. HID, communications, mass storage etc.). This allows many devices to use

class drivers preventing the need to write specific drivers for your device. The **iInterface** item allows for a string description of the interface.

You can see below an example of a structure containing the interface descriptor fields:

```
typedef struct __attribute__ ((packed))
{
    uint8_t bLength;            // Size of Descriptor in Bytes (9 Bytes)
    uint8_t bDescriptorType;    // Interface Descriptor (0x04)
    uint8_t bInterfaceNumber;   // Number of Interface
    uint8_t bAlternateSetting;  // Value used to select alternative setting
    uint8_t bNumEndPoints;      // Number of Endpoints used for this interface
    uint8_t bInterfaceClass;    // Class Code (Assigned by USB Org)
    uint8_t bInterfaceSubClass; // Subclass Code (Assigned by USB Org)
    uint8_t bInterfaceProtocol; // Protocol Code (Assigned by USB Org)
    uint8_t iInterface;         // Index of String Descriptor Describing this
interface

} USB_INTERFACE_DESCRIPTOR;
```

USB Endpoint Descriptor

Endpoint descriptors are used to describe endpoints other than endpoint zero. Endpoint zero is always assumed to be a control endpoint and is configured before any descriptors are even requested. The host will use the information returned from these descriptors to determine the bandwidth requirements of the bus. You can see below an example of a structure that contains all the endpoint descriptor fields:

```
typedef struct __attribute__ ((packed))
{
    uint8_t bLength;            // Size of Descriptor in Bytes (7 bytes)
    uint8_t bDescriptorType;    // Endpoint Descriptor (0x05)
    uint8_t bEndpointAddress;   // Endpoint Address.Bits 0..3b Endpoint Number. Bits
4..6b Reserved.Set to Zero.Bits 7 Direction 0 = Out, 1 = In
    uint8_t bmAttributes        // Transfer type
    uint16_t wMaxPacketSize;    // Maximum Packet Size this endpoint can send or
receive
    uint8_t  bInterval;         // Interval for polling endpoint data transfers

} USB_ENDPOINT_DESCRIPTOR;
```

The **bEndpointAddress** indicates what endpoint this descriptor is describing.

The **bmAttributes** specifies the transfer type. This can either be Control, Interrupt, Isochronous or Bulk Transfers. If an Isochronous endpoint is specified, additional attributes can be selected such as the synchronisation and usage types. **Bits 0..1** are the transfer type: 00 = Control, 01 =

Isochronous, 10 = Bulk, 11 = Interrupt. **Bits 2..7** are reserved. If the endpoint is Isochronous **Bits 3..2** = Synchronisation Type (Iso Mode): 00 = No Synchronization, 01 = Asynchronous, 10 = Adaptive, 11 = Synchronous. **Bits 5..4** = Usage Type (Iso Mode): 00 = Data Endpoint, 01 = Feedback Endpoint, 10 = Explicit Feedback Data Endpoint, 11 = Reserved.

The **wMaxPacketSize** item indicates the maximum payload size for this endpoint.

The **bInterval** item is used to specify the polling interval of endpoint data transfers. Ignored for Bulk and Control Endpoints. The units are expressed in frames, thus this equates to either 1ms for low/full speed devices and 125us for high speed devices.

USB String Descriptors

String descriptors (USB_STRING_DESCRIPTOR) are optional and add human readable information to the other descriptors. If a device does not support string descriptors, all references to string descriptors within device, configuration, and interface descriptors must be set to zero.

String descriptors are encoded in Unicode so that multiple languages can be supported with a single product. When requesting a string descriptor, the requester specifies the desired language using a 16-bit language ID (LANGID) defined by the USB-IF (refer to Language Identifiers (LANGIDs)). String index zero is used for all languages and returns a string descriptor that contains an array of two-byte LANGID codes supported by the device.

The array of LANGID codes is not NULL-terminated. The size of the array (in byte) is computed by subtracting two from the value of the first byte to the descriptor.

Offset	Field	Type	Size	Value	Description
0	bLength	uint8_t	N + 2	Number	Size of this descriptor in bytes.
1	bDescriptorType	uint8_t	1	Constant	String Descriptor Type
2	wLANGID[0]	uint8_t	2	Number	LANGID code zero (for example 0x0407 German (Standard)).
...	...	...	...	...	...
N	wLANGID[x]	uint8_t	2	Number	LANGID code zero x (for example 0x0409 English (United States)).

The UNICODE string descriptor is not NULL-terminated. The string length is computed by subtracting two from the value of the first byte of the descriptor.

Offset	Field	Type	Size	Value	Description
0	bLength	uint8_t	1	Number	Size of this descriptor in bytes.
1	bDescriptorType	uint8_t	1	Constant	String Descriptor Type
2	bString	uint8_t	N	Number	UNICODE encoded string.

USB HID Descriptor

The USB HID Device Class supports devices that are used by humans to control the operation of computer systems. The HID class of devices include a wide variety of human interface, data indicator, and data feedback devices with various types of output directed to the end user. Some common examples of HID class devices include:

- Keyboards
- Pointing devices such as a standard mouse, joysticks, and trackballs
- Front-panel controls like knobs, switches, buttons, and sliders
- Controls found on telephony, gaming or simulation devices such as steering wheels, rudder pedals, and dial pads
- Data devices such as bar-code scanners, thermometers, analyzers

The following descriptors are required in an USB HID Device:

- Standard Device Descriptor
- Standard Configuration Descriptor
- Standard Interface Descriptor for the HID Class
- Class-Specific HID Descriptor
- Standard Endpoint Descriptor for Interrupt IN endpoint
- Class-Specific Report Descriptor

The Class-Specific HID descriptor looks like this:

```
typedef struct __attribute__((packed))
{
    uint8_t     bLength;
    uint8_t     bDescriptorType;
    uint16_t    bcdHID;
    uint8_t     bCountryCode;
    uint8_t     bNumDescriptors;
    uint8_t     bReportDescriptorType;
    uint16_t    wItemLength;

} USB_HID_DESCRIPTOR;
```

The **bLength** item describes the size of the HID descriptor. It can vary depending on the number of subordinate descriptors, such as report descriptors, that are included in this HID configuration definition.

The **bDescriptorType** 0x21 value is the constant one-byte designator for device descriptors and should be common to all HID descriptors.

The two-byte **bcdHID** item tells the host which version of the HID class specification the device follows. USB specification requires that this value be formatted as a binary coded decimal digit, meaning that the upper and lower nibbles of each byte represent the number '0'...9'. For example, 0x0101 represents the number 0101, which equals a revision number of 1.01 with an implied decimal point.

If the device was designed to be localized to a specific country, the **bCountryCode** item tells the host which country. Setting the item to 0x00 tells the host that the device was not designed to be localized to any country.

The **bNumDescriptors** item tells the host how many report descriptors are contained in this HID configuration. The following two-byte pairs of items describe each contained report descriptor.

The **bReportDescriptorType** item describes the first descriptor which will follow the transfer of this HID descriptor. For example, if the value is "0x22" indicates that the descriptor to follow is a report descriptor.

The **wItemLength** item tells the host the size of the descriptor that is described in the preceding item.

The **HID report descriptor** is a hard coded array of bytes that describe the device's data packets. This includes: how many packets the device supports, how large are the packets, and the purpose of each byte and bit in the packet. For example, a keyboard with a calculator program button can tell the host that the button's pressed/released state is stored as the 2nd bit in the 6th byte in data packet number 4.

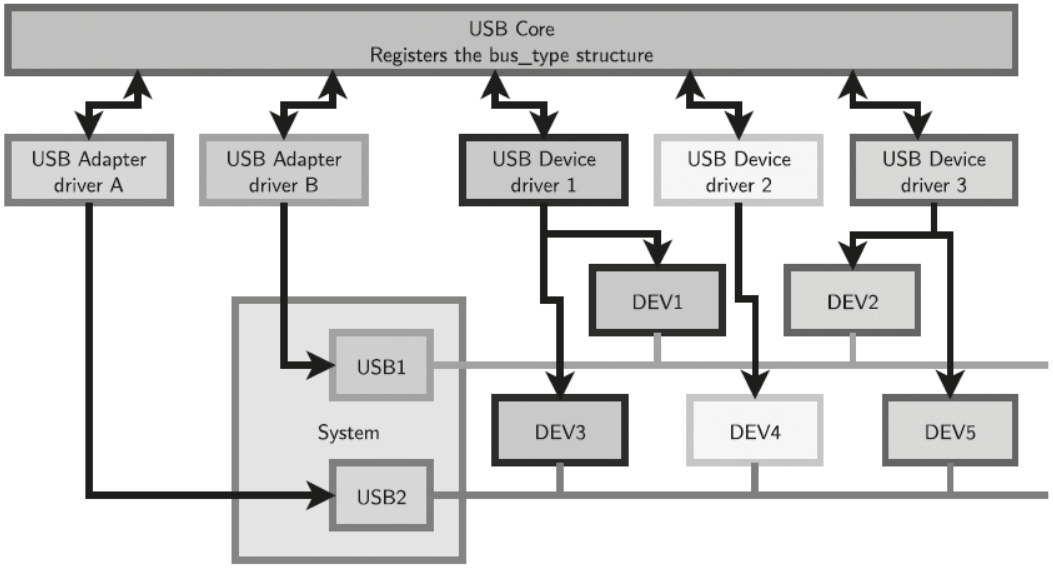
The Linux USB Subsystem

Kernel developers added USB support to Linux early in the 2.2 kernel series and have been developing it further since then. Besides support for each new generation of USB, various host controllers gained support, new drivers for peripherals have been added and advanced features for latency measurement and improved power management introduced.

Linux can run inside USB devices as well as on the hosts that control the devices. But USB device drivers running inside those peripherals don't do the same things as the ones running inside hosts, so they've been given a different name: **gadget drivers**. We will focus on this chapter in the development of Linux USB device drivers running on hosts.

In Linux there exists a subsystem called "The USB Core" with a specific API to support USB devices and host controllers. Its purpose is to abstract all hardware or device dependent parts by

defining a set of data structures, macros and functions. Host-side drivers for USB devices talk to these "usbcore" APIs. There are two set of APIs, one is intended for general-purpose USB device drivers (the ones that will be developed through this chapter), and the other is for drivers that are part of the core. Such core drivers include the hub driver (which manages trees of USB devices) and several different kinds of USB host adapter drivers, which control individual busses. The following image shows an example of a Linux USB Subsystem:



The Linux USB API supports synchronous calls for control and bulk messages. It also supports asynchronous calls for all kinds of data transfer by using request structures called "URBs" (USB Request Blocks).

The only host-side drivers that actually touch hardware (reading/writing registers, handling IRQs, and so on) are the Host Controller Devices (HCDs) drivers. In theory, all HCDs provide the same functionality through the same API. In practice, that's becoming more true, but there are still differences that crop up especially with fault handling on the less common controllers. Different controllers don't necessarily report the same aspects of failures, and recovery from faults (including software-induced ones like unlinking an URB) isn't yet fully consistent.

Further information about USB Core and Host Controller APIs can be found at:

http://heim.ifi.uio.no/~knuto/kernel/4.14/driver-api/usb/usb.html#usb-core-apis
http://heim.ifi.uio.no/~knuto/kernel/4.14/driver-api/usb/usb.html#host-controller-apis

The main focus of this chapter is the development of Linux Host USB device drivers. All the sections that follow are related to the development of this type of drivers.

Writing Linux USB Device Drivers

In the following labs, you will develop several USB device drivers through which you will understand the basic framework of a Linux USB device driver. But before you proceed with the labs, you need to familiarize yourselves with the main USB data structures and functions. The following sections will explain these structures and functions in detail.

USB Device Driver Registration

The first thing a Linux USB device driver needs to do is register itself with the Linux USB core, giving it some information about which devices the driver supports and which functions to call when a device supported by the driver is inserted or removed from the system. All of this information is passed to the USB core in the usb_driver structure. See below the struct usb_driver definition for an USB seven segment driver located at /linux/drivers/misc/usbsevseg.c:

```
static struct usb_driver sevseg_driver = {
   .name =               "usbsevseg",
   .probe =              sevseg_probe,
   .disconnect =         sevseg_disconnect,
   .suspend =            sevseg_suspend,
   .resume =             sevseg_resume,
   .reset_resume =       sevseg_reset_resume,
   .id_table =           id_table,
};
```

The variable name is a string that describes the driver. It is used in informational messages printed to the system log. The probe() and disconnect() hotplugging callbacks are called when a device that matches the information provided in the id_table variable is either seen or removed.

The probe() function is called by the USB core into the driver to see if the driver is willing to manage a particular interface on a device. If it is, probe() returns zero and uses usb_set_intfdata() to associate driver specific data with the interface. It may also use usb_set_interface() to specify the appropriate altsetting. If unwilling to manage the interface, return -ENODEV, if genuine IO errors occurred, an appropriate negative errno value.

```
int (* probe) (struct usb_interface *intf,const struct usb_device_id *id);
```

The disconnect() callback is called when the interface is no longer accessible, usually because its device has been (or is being) disconnected or the driver module is being unloaded:

```
void disconnect(struct usb device *dev, void *drv context);
```

In the struct usb_driver there are defined some power management (PM) callbacks:

- **suspend**: called when the device is going to be suspended.
- **resume**: called when the device is being resumed.
- **reset_resume**: called when the suspended device has been reset instead of being resumed.

And there are also defined some device level operations:

- **pre_reset**: called when the device is about to be reset.
- **post_reset**: called after the device has been reset.

The USB device drivers use ID table to support hotplugging. The pointer variable id_table included in the usb_driver structure points to an array of structures of type usb_device_id that announce the devices that the USB device driver supports. Most drivers use the USB_DEVICE() macro to create usb_device_id structures. These structures are registered to the USB core by using the MODULE_DEVICE_TABLE(usb, xxx) macro. The following lines of code included in the /linux/drivers/misc/usbsevseg.c driver create and register an USB device to the USB core:

```
#define VENDOR_ID       0x0fc5
#define PRODUCT_ID      0x1227

/* table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
   { USB_DEVICE(VENDOR_ID, PRODUCT_ID) },
   { },
};
MODULE_DEVICE_TABLE(usb, id_table);
```

The usb_driver structure is registered to the bus core by using the module_usb_driver() function:

```
module_usb_driver(sevseg_driver);
```

Linux Host-Side Data Types

USB device drivers actually bind to interfaces, not devices. Think of them as "interface drivers", though you may not see many devices where the distinction is important. Most USB devices are simple, with only one function, one configuration, one interface, and one alternate setting. The USB interface is represented by the usb_interface structure. This is what the USB core passes to the USB driver´s probe() function when this callback function is being called.

```
struct usb_interface {
  struct usb_host_interface * altsetting;
  struct usb_host_interface * cur_altsetting;
```

```
    unsigned num_altsetting;
    struct usb_interface_assoc_descriptor * intf_assoc;
    int minor;
    enum usb_interface_condition condition;
    unsigned sysfs_files_created:1;
    unsigned ep_devs_created:1;
    unsigned unregistering:1;
    unsigned needs_remote_wakeup:1;
    unsigned needs_altsetting0:1;
    unsigned needs_binding:1;
    unsigned resetting_device:1;
    unsigned authorized:1;
    struct device dev;
    struct device * usb_dev;
    atomic_t pm_usage_cnt;
    struct work_struct reset_ws;
};
```

These are the main members of the usb_interface structure:

- **altsetting**: array of usb_host_interface structures, one for each alternate setting that may be selected. Each one includes a set of endpoint configurations. They will be in no particular order. The usb_host_interface structure for each alternate setting allows to access the usb_endpoint_descriptor structure for each of its endpoints:

  ```
  interface->altsetting[i]->endpoint[j]->desc
  ```

- **cur_altsetting**: the current altsetting.
- **num_altsetting**: number of altsettings defined.

Each interface may have alternate settings. The initial configuration of a device sets altsetting 0, but the device driver can change that setting by using usb_set_interface(). Alternate settings are often used to control the use of periodic endpoints, such as by having different endpoints use different amounts of reserved USB bandwidth. All standards-conformant USB devices that use isochronous endpoints will use them in non-default settings.

The struct usb_host_interface includes an array of usb_host_endpoint structures.

```
/* host-side wrapper for one interface setting's parsed descriptors */
struct usb_host_interface {
    struct usb_interface_descriptor     desc;

    int extralen;
    unsigned char *extra;   /* Extra descriptors */

    /* array of desc.bNumEndpoints endpoints associated with this
     * interface setting.  these will be in no particular order.
```

```
     */
    struct usb_host_endpoint *endpoint;

    char *string;         /* iInterface string, if present */
};
```

Each usb_host_endpoint structure includes a struct usb_endpoint_descriptor.

```
struct usb_host_endpoint {
    struct usb_endpoint_descriptor      desc;
    struct usb_ss_ep_comp_descriptor    ss_ep_comp;
    struct usb_ssp_isoc_ep_comp_descriptor      ssp_isoc_ep_comp;
    struct list_head             urb_list;
    void                         *hcpriv;
    struct ep_device             *ep_dev;       /* For sysfs info */

    unsigned char *extra;   /* Extra descriptors */
    int extralen;
    int enabled;
    int streams;
};
```

The usb_endpoint_descriptor structure contains all the USB-specific data announced by the device itself.

```
struct usb_endpoint_descriptor {
    __u8  bLength;
    __u8  bDescriptorType;

    __u8  bEndpointAddress;
    __u8  bmAttributes;
    __le16 wMaxPacketSize;
    __u8  bInterval;

    /* NOTE:  these two are _only_ in audio endpoints. */
    /* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
    __u8  bRefresh;
    __u8  bSynchAddress;
} __attribute__ ((packed));
```

You can use the following code to obtain the IN and OUT endpoint addresses from the IN and OUT endpoint descriptors, which are included in the current altsetting of the USB interface:

```
struct usb_host_interface *altsetting = intf->cur_altsetting;
int ep_in, ep_out;

/* there are two usb_host_endpoint structures in this interface altsetting.Each usb_
host_endpoint structure contains a usb_endpoint_descriptor */
ep_in = altsetting->endpoint[0].desc.bEndpointAddress;
ep_out = altsetting->endpoint[1].desc.bEndpointAddress;
```

USB Request Block (URB)

Any communication between the host and device is done asynchronously by using USB Request Blocks (urbs).

- An URB consists of all relevant information to execute any USB transaction and deliver the data and status back.
- Execution of an URB is inherently an asynchronous operation, i.e. the usb_submit_urb() call returns immediately after it has successfully queued the requested action.
- Transfers for one URB can be canceled with usb_unlink_urb() at any time.
- Each URB has a completion handler, which is called after the action has been successfully completed or canceled. The URB also contains a context-pointer for passing information to the completion handler.
- Each endpoint for a device logically supports a queue of requests. You can fill that queue, so that the USB hardware can still transfer data to an endpoint while your driver handles completion of another. This maximizes use of USB bandwidth, and supports seamless streaming of data to (or from) devices when using periodic transfer modes.

These are some fields of the struct urb:

```
struct urb
{
      // (IN) device and pipe specify the endpoint queue
      struct usb_device *dev;         // pointer to associated USB device
      unsigned int pipe;              // endpoint information

      unsigned int transfer_flags;    // URB_ISO_ASAP, URB_SHORT_NOT_OK, etc.

      // (IN) all urbs need completion routines
      void *context;                  // context for completion routine
      usb_complete_t complete;        // pointer to completion routine

      // (OUT) status after each completion
      int status;                     // returned status

      // (IN) buffer used for data transfers
      void *transfer_buffer;          // associated data buffer
      u32 transfer_buffer_length;     // data buffer length
      int number_of_packets;          // size of iso_frame_desc

      // (OUT) sometimes only part of CTRL/BULK/INTR transfer_buffer is used
      u32 actual_length;              // actual data buffer length
```

```
        // (IN) setup stage for CTRL (pass a struct usb_ctrlrequest)
         unsigned char *setup_packet;    // setup packet (control only)

        // Only for PERIODIC transfers (ISO, INTERRUPT)
        // (IN/OUT) start_frame is set unless URB_ISO_ASAP isn't set
         int start_frame;               // start frame
         int interval;                  // polling interval

         // ISO only: packets are only "best effort"; each can have errors
         int error_count;               // number of errors
         struct usb_iso_packet_descriptor iso_frame_desc[0];
};
```

The USB driver must create a "pipe" using values from the appropriate endpoint descriptor in an interface that it's claimed.

URBs are allocated by calling usb_alloc_urb():

```
struct urb *usb_alloc_urb(int isoframes, int mem_flags)
```

Return value is a pointer to the allocated URB, 0 if allocation failed. The parameter isoframes specifies the number of isochronous transfer frames you want to schedule. For CTRL/BULK/INT, use 0. The mem_flags parameter holds standard memory allocation flags, letting you control (among other things) whether the underlying code may block or not.

To free an URB, use usb_free_urb():

```
void usb_free_urb(struct urb *urb)
```

Interrupt transfers are periodic, and happen in intervals that are powers of two (1, 2, 4 etc) units. Units are frames for full and low speed devices, and microframes for high speed ones. You can use the usb_fill_int_urb() macro to fill INT transfer fields. When the write urb is filled up with the proper information by using the usb_fill_int_urb() function, you should point the urb's completion callback to call your own callback function. This function is called when the urb is finished by the USB subsystem. The callback function is called in interrupt context, so caution must be taken not to do very much processing at that time. The usb_submit_urb() call modifies urb->interval to the implemented interval value that is less than or equal to the requested interval value.

An URB is submitted by using the function usb_submit_urb():

```
int usb_submit_urb(struct urb *urb, int mem_flags)
```

The mem_flags parameter, such as GFP_ATOMIC, controls memory allocation, such as whether the lower levels may block when memory is tight. It immediately returns, either with status 0 (request queued) or some error code, usually caused by the following:

- Out of memory (-ENOMEM)
- Unplugged device (-ENODEV)
- Stalled endpoint (-EPIPE)
- Too many queued ISO transfers (-EAGAIN)
- Too many requested ISO frames (-EFBIG)
- Invalid INT interval (-EINVAL)
- More than one packet for INT (-EINVAL)

After submission, urb->status is -EINPROGRESS; however, you should never look at that value except in your completion callback.

There are two ways to cancel an URB you've submitted but which hasn't been returned to your driver yet. For an asynchronous cancel, call usb_unlink_urb():

```
int usb_unlink_urb(struct urb *urb)
```

It removes the urb from the internal list and frees all allocated HW descriptors. The status is changed to reflect unlinking. Note that the URB will not normally have finished when usb_unlink_urb() returns; you must still wait for the completion handler to be called.

To cancel an URB synchronously, call usb_kill_urb():

```
void usb_kill_urb(struct urb *urb)
```

It does everything usb_unlink_urb() does, and in addition it waits until after the URB has been returned and the completion handler has finished.

The completion handler is of the following type:

```
typedef void (*usb_complete_t)(struct urb *)
```

In the completion handler, you should have a look at urb->status to detect any USB errors. Since the context parameter is included in the URB, you can pass information to the completion handler.

LAB 13.1: USB HID Device Application

In this first USB lab, you will learn how to create a fully functional USB HID device and how to send and receive data by using HID reports. For this lab, you are going to use the Curiosity PIC32MX470 Development Board:

https://www.microchip.com/DevelopmentTools/ProductDetails/dm320103#additional-summary

The Curiosity PIC32 MX470 Development Board features PIC32MX Series (PIC32MX470512H) with a 120MHz CPU, 512KB Flash, 128KB RAM , Full Speed USB and multiple expansion options.

The Curiosity Development Board includes an integrated programmer/debugger, excellent user experience options with Multiple LED's, RGB LED and a switch. Each board provides two MikroBus® expansion sockets from MicroElektronika , I/O expansion header and a Microchip X32 header to enable customers seeking accelerated application prototype development. The board is fully integrated with Microchip's MPLAB® X IDE and into PIC32's powerful software framework, MPLAB ® Harmony that a provides flexible and modular interface to application development, a rich set of inter-operable software stack (TCP-IP, USB) and easy to use features.

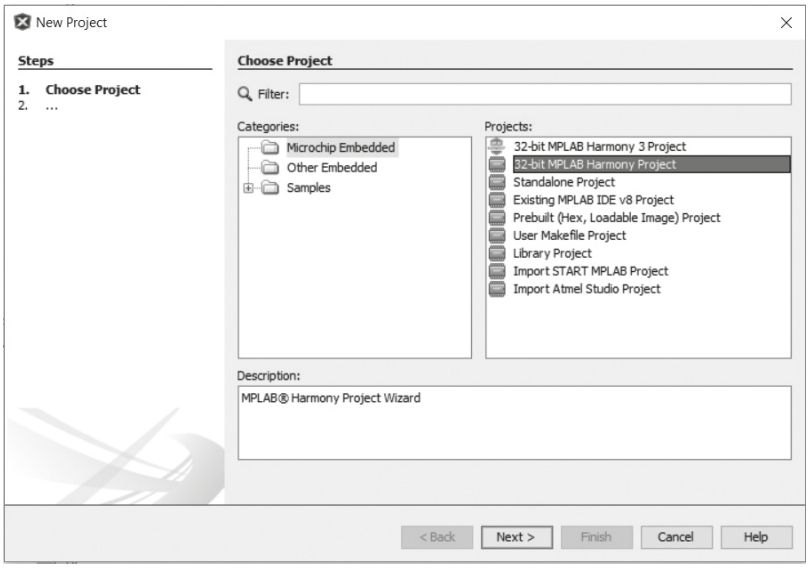
These are the SW and HW requirements for the lab:

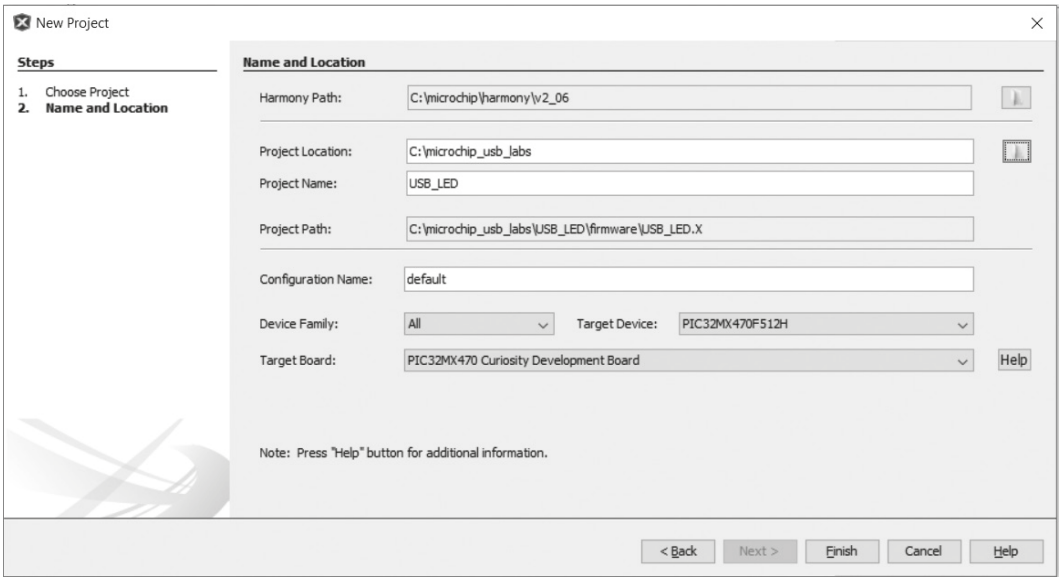
- **Development Environment**: MPLAB® X IDE v5.10
- **C Compiler**: MPLAB® XC32 v2.15
- **Software Tools**: MPLAB® Harmony Integrated Software Framework v2.06. GenericHIDSimpleDemo application ("hid_basic" example of Harmony)
- **Hardware Tools**: Curiosity PIC32MX470 Development Board (dm320103)

The objective of this lab is using the MPLAB® Harmony Configurator Tool, create a MPLAB X project and write the code to make an USB Device, so that it can be enumerated as a HID device and communicate with the Linux USB host driver that you will develop in the following lab.

STEP 1: Create a New Project

Create an empty 32-bit MPLAB Harmony Project, named USB_LED, for the Curiosity development board. Save the project in the following folder that was previously created: C:\microchip_usb_labs.



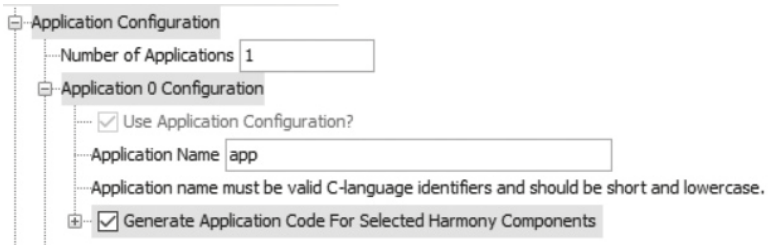


STEP 2: Configure Harmony

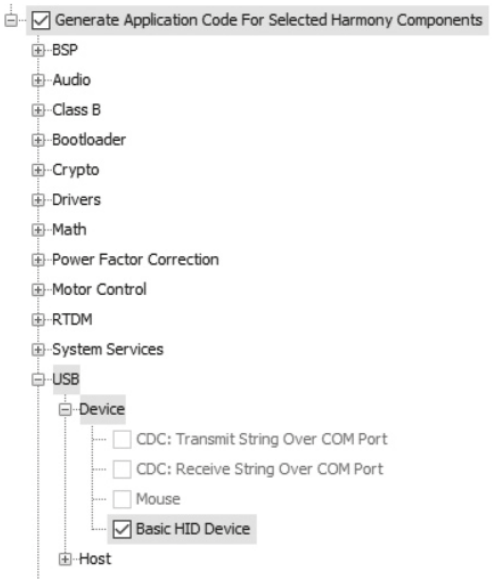
Launch the MPLAB Harmony Configurator plugin and click on Tools->Embedded->MPLAB Harmony Configurator.

Select your demo board enabling the BSP (Board Support Package):

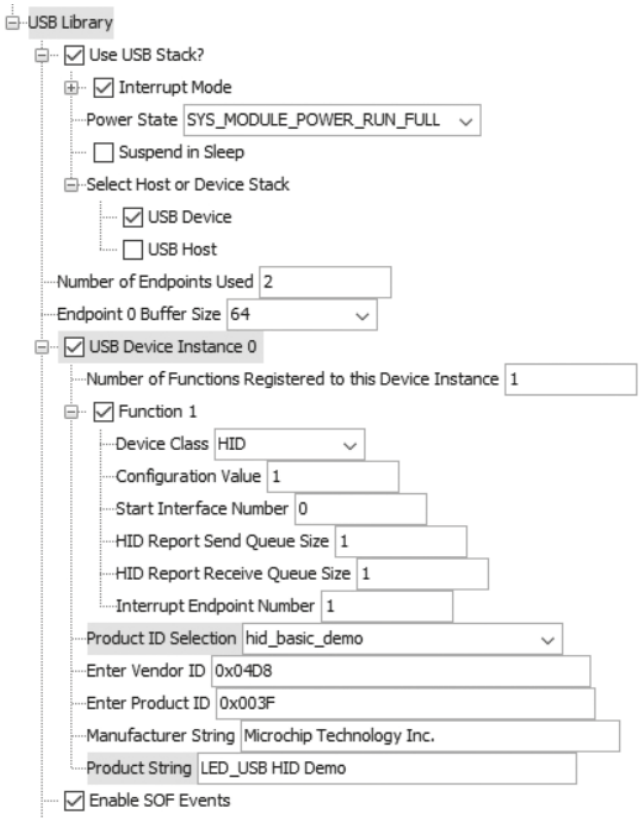
Enable the Generate Application Code For Selected Harmony Components:



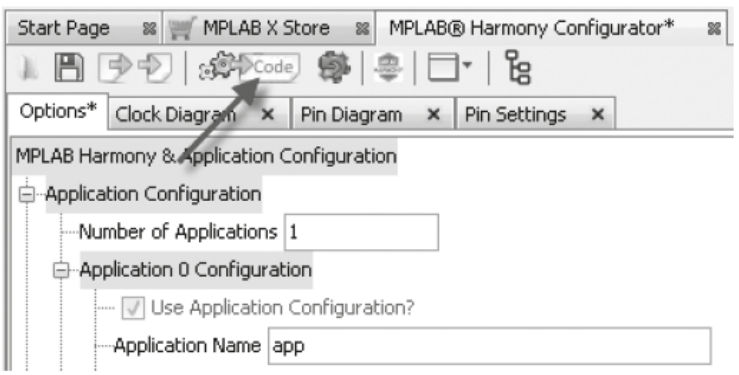
Select the Basic HID Device demo template:



In the USB Library option of Harmony Framework Configuration select hid_basic_demo as Product ID Selection. Select also the Vendor ID, Product ID, Manufacturer String and Product String as shown in the screen capture below. You have to select an USB Device stack. The USB device will have one control endpoint (ep0) and one interrupt endpoint (composed of IN and OUT endpoints), so you will have to write the value two in the Number of Endpoints Used field. There will be only one configuration and one interface associated with the device.



Generate the code, save the modified configuration, and generate the project:



STEP 3: Modify the Generated Code

Typically, the HID class is used to implement human interface products, such as mice and keyboards. The HID protocol, is however, quite flexible, and can be adapted and used to send/receive general purpose data to/from an USB device.

In the following two labs, you will see how to use the USB protocol for basic general purpose USB data transfer. You will develop Linux USB host drivers that send USB commands to the PIC32MX USB HID device to toggle three LEDs (LED1, LED2, LED3) included in the PIC32MX Curiosity board. The PIC32MX USB HID device will also check the value of the user button (S1) and will reply to the host with a packet that contains the value of that switch.

In this lab, you have to implement the following in the USB device side:

- **Toggle LED(s)**: The Linux USB host driver sends a report to the HID device. The first byte of the report can be 0x01, 0x02, or 0x03. The HID device must toggle LED1 when it receives 0x01, LED2 when it receives 0x02 and LED3 when it receives 0x03 in the report .
- **Get Pushbutton State**: The Linux USB host driver sends a report to the HID Device. The first byte of this report is 0x00. The HID device must reply with another report, where the first byte is the status of the S1 button ("0x00" pressed, "0x01" not pressed)

By examining the app.c code generated by MPLAB Harmony Configurator, the template is expecting you to implement your USB state machine inside the Function USB_Task(). The state machine you need to implement will be executed if the HID device is configured; if the HID device is de-configured, the USB state machine needs to return to the INIT state.

After initialization, the HID device needs to wait for a command from the host; scheduling a read request will enable the HID device to receive a report. The state machine needs to wait for the host to send the report; after receiving the report, the application needs to check the first byte of the report. If this byte is 0x01, 0x02 or 0x03, then LED1, LED2 and LED3 must be toggled. If the first byte is 0x00, a response report with the switch status must be sent to the host and then a new read must be scheduled.

STEP 4: Declare the USB State Machine States

To create the USB State Machine, you need to declare an enumeration type (e.g. USB_STATES) containing the labels for the four states, needed to implement the state machine. (e.g. USB_STATE_INIT, USB_STATE_WAITING_FOR_DATA, USB_STATE_SCHEDULE_READ, USB_STATE_SEND_REPORT). Find the section Type Definitions in app.h file and declare the enumeration type.

```
typedef enum
{
   /* Application's state machine's initial state. */
   APP_STATE_INIT=0,
   APP_STATE_SERVICE_TASKS,

   /* TODO: Define states used by the application state machine. */

} APP_STATES;

/* Declare the USB State Machine states */
typedef enum
{
   /* Application's state machine's initial state. */
   USB_STATE_INIT=0,
    USB_STATE_WAITING_FOR_DATA,
    USB_STATE_SCHEDULE_READ,
    USB_STATE_SEND_REPORT

} USB_STATES;
```

STEP 5: Add New Members to APP_DATA Type

The APP_DATA structure type already contains members needed for the Application State Machine and the enumeration process (state, handleUsbDevice, usbDeviceIsConfigured, etc.); you need to add the members you will use to send and receive HID reports.

Find the APP_DATA structure type in app.h file and add the following members:

- a member to store the USB State Machine status
- two pointers to buffer (one for data received and one for data to send)
- two HID transfer handles (one for reception transfer, one for the transmission transfer)
- two flags to indicate the state of the ongoing transfer (one for reception, one for transmission transfer)

```
typedef struct
{
    /* The application's current state */
    APP_STATES state;

    /* TODO: Define any additional data used by the application. */

    /*
     * USB variables used by the HID device application:
     *
```

```
     *     handleUsbDevice          : USB Device driver handle
     *     usbDeviceIsConfigured    : If true, USB Device is configured
     *     activeProtocol           : USB HID active Protocol
     *     idleRate                 : USB HID current Idle
     */
    USB_DEVICE_HANDLE               handleUsbDevice;
    bool                            usbDeviceIsConfigured;
    uint8_t                         activeProtocol;
    uint8_t                         idleRate;

    /* Add new members to APP_DATA type */
    /* USB_Task's current state */
    USB_STATES stateUSB;

    /* Receive data buffer */
    uint8_t * receiveDataBuffer;

    /* Transmit data buffer */
    uint8_t * transmitDataBuffer;

    /* Send report transfer handle*/
    USB_DEVICE_HID_TRANSFER_HANDLE txTransferHandle;

    /* Receive report transfer handle */
    USB_DEVICE_HID_TRANSFER_HANDLE rxTransferHandle;

    /* HID data received flag*/
    bool hidDataReceived;

    /* HID data transmitted flag */
    bool hidDataTransmitted;

} APP_DATA;
```

STEP 6: Declare the Reception and Transmission Buffers

To schedule a report receive or a report send request, you need to provide a pointer to a buffer to store the received data and the data that has to be transmitted. Find the section Global Data Definitions in app.c file and declare two 64 byte buffers.

```
APP_DATA appData;

/* Declare the reception and transmission buffers */
uint8_t receiveDataBuffer[64] __attribute__((aligned(16)));
uint8_t transmitDataBuffer[64] __attribute__((aligned(16)));
```

STEP 7: Initialize the New Members

In Step 5, you added some new members to APP_DATA structure type; those members need to be initialized and some of them need to be initialized just once in the APP_Initialize() function.

Find the APP_Initialize() function in app.c file, and add the code to initialize the USB State Machine state member and the two buffer pointers; the state variable needs to be set to the initial state of the USB State Machine. The two pointers need to point to the corresponding buffers you declared in Step 6.

The other members will be initialized just before their use.

```
void APP_Initialize ( void )
{
    /* Place the App state machine in its initial state. */
    appData.state = APP_STATE_INIT;

    /* Initialize USB HID Device application data */
    appData.handleUsbDevice       = USB_DEVICE_HANDLE_INVALID;
    appData.usbDeviceIsConfigured = false;
    appData.idleRate              = 0;

    /* Initialize USB Task State Machine appData members */
    appData.receiveDataBuffer = &receiveDataBuffer[0];
    appData.transmitDataBuffer = &transmitDataBuffer[0];
    appData.stateUSB = USB_STATE_INIT;

}
```

STEP 8: Handle the Detach

In the Harmony version we are using, USB_DEVICE_EVENT_DECONFIGURED and USB_DEVICE_EVENT_RESET events are not passed to the Application USB Device Event Handler Function. So the usbDeviceIsConfigured flag of appData structure needs to be set as false inside the USB_DEVICE_EVENT_POWER_REMOVED event.

Find the power removed case (USB_DEVICE_EVENT_POWER_REMOVED), in the APP_USBDeviceEventHandler() function, in app.c file and set the member usbDeviceIsConfigured of appData structure to false.

```
case USB_DEVICE_EVENT_POWER_REMOVED:
   /* VBUS is not available any more. Detach the device. */
   /* STEP 8: Handle the detach */
   USB_DEVICE_Detach(appData.handleUsbDevice);
   appData.usbDeviceIsConfigured = false;
   /* This is reached from Host to Device */
   break;
```

STEP 9: Handle the HID Events

The two flags you declared in Step 5 will be used by the USB State Machine to check the status of the previous report receive or transmit transaction. The status of those two flags need to be updated when the two HID events (report sent and report received) are passed to the Application HID Event Handler Function. You need to be sure that the event is related to the request you made and, for this purpose, you can compare the transfer handle of the request with the transfer handle available in the event: if they match, the event is related to the ongoing request.

Find the APP_USBDeviceHIDEventHandler() function in app.c file, add a local variable to cast the eventData parameter and update the two flags, one in the report received event, one in the report sent event; don't forget to check if the transfer handles are matching before setting the flag to true. To match the transfer handle you need to cast the eventData parameter to the USB Device HID Report Event Data Type; there are two events and two types, one for report received and one for report sent.

```
static void APP_USBDeviceHIDEventHandler
(
    USB_DEVICE_HID_INDEX hidInstance,
    USB_DEVICE_HID_EVENT event,
    void * eventData,
    uintptr_t userData
)
{
    APP_DATA * appData = (APP_DATA *)userData;

    switch(event)
    {
        case USB_DEVICE_HID_EVENT_REPORT_SENT:
        {
            /* This means a Report has been sent.  We are free to send next
             * report. An application flag can be updated here. */

            /* Handle the HID Report Sent event */
            USB_DEVICE_HID_EVENT_DATA_REPORT_SENT * report =
                    (USB_DEVICE_HID_EVENT_DATA_REPORT_SENT *)eventData;
            if(report->handle == appData->txTransferHandle )
            {
                // Transfer progressed.
                appData->hidDataTransmitted = true;
            }
            break;
        }
        case USB_DEVICE_HID_EVENT_REPORT_RECEIVED:
        {
```

```
                /* This means Report has been received from the Host. Report
                 * received can be over Interrupt OUT or Control endpoint based on
                 * Interrupt OUT endpoint availability. An application flag can be
                 * updated here. */

                 /* Handle the HID Report Received event */
                USB_DEVICE_HID_EVENT_DATA_REPORT_RECEIVED * report =
                        (USB_DEVICE_HID_EVENT_DATA_REPORT_RECEIVED *)eventData;
                if(report->handle == appData->rxTransferHandle )
                {
                    // Transfer progressed.
                    appData->hidDataReceived = true;
                }
                break;
            }

        [...]

    }
```

STEP 10: Create the USB State Machine

The Basic HID Device template that was used to generate the code expects the USB State machine to be placed inside the USB_Task() function; that state machine will be executed until the usbDeviceIsConfigured member of appData structure, is true.

When the USB cable is unplugged, the state machine is no longer executed but you need to reset it to the initial state to be ready for the next USB connection.

Find the if(appData.usbDeviceIsConfigured) statement of USB_Task() function in app.c file and add the else statement to set the USB State Machine state member of the appData structure to its initial state (e.g. USB_STATE_INIT).

Inside the if statement of the USB_Task() function, you can place the requested state machine; you can create it using a switch statement with four cases, one for each state you declared in the enumeration type you defined in Step 4. Find the if(appData.usbDeviceIsConfigured) statement of the USB_Task() function and add a switch statement for the USB State Machine state member of the appData structure and a case for each entry of the enumeration type of that state member.

Inside the initialization state of the switch statement add the code to set the transmission flag to true and the two transfer handles to invalid (USB_DEVICE_HID_TRANSFER_HANDLE_INVALID), set the USB State Machine state member of appData structure to the state that schedules a receive request (e.g. USB_STATE_SCHEDULE_READ).

```
static void USB_Task (void)
{
    if(appData.usbDeviceIsConfigured)
    {
        /* Write USB HID Application Logic here. Note that this function is
         * being called periodically the APP_Tasks() function. The application
         * logic should be implemented as state machine. It should not block */

        switch (appData.stateUSB)
        {
            case USB_STATE_INIT:

                appData.hidDataTransmitted = true;
                appData.txTransferHandle = USB_DEVICE_HID_TRANSFER_HANDLE_INVALID;
                appData.rxTransferHandle = USB_DEVICE_HID_TRANSFER_HANDLE_INVALID;
                appData.stateUSB = USB_STATE_SCHEDULE_READ;

                break;

            case USB_STATE_SCHEDULE_READ:

                appData.hidDataReceived = false;
                USB_DEVICE_HID_ReportReceive (USB_DEVICE_HID_INDEX_0,
                    &appData.rxTransferHandle, appData.receiveDataBuffer, 64 );
                appData.stateUSB = USB_STATE_WAITING_FOR_DATA;

                break;

            case USB_STATE_WAITING_FOR_DATA:

                if( appData.hidDataReceived )
                {
                    if (appData.receiveDataBuffer[0]==0x01)
                    {
                        BSP_LED_1Toggle();
                        appData.stateUSB = USB_STATE_SCHEDULE_READ;
                    }
                    else if (appData.receiveDataBuffer[0]==0x02)
                    {
                        BSP_LED_2Toggle();
                        appData.stateUSB = USB_STATE_SCHEDULE_READ;
                    }
                    else if (appData.receiveDataBuffer[0]==0x03)
                    {
                        BSP_LED_3Toggle();
                        appData.stateUSB = USB_STATE_SCHEDULE_READ;
                    }
                    else if (appData.receiveDataBuffer[0]==0x00)
```

```
                    {
                        appData.stateUSB = USB_STATE_SEND_REPORT;
                    }
                    else
                    {
                        appData.stateUSB = USB_STATE_SCHEDULE_READ;
                    }
                }

                break;

            case USB_STATE_SEND_REPORT:

                if(appData.hidDataTransmitted)
                {
                    if( BSP_SwitchStateGet(BSP_SWITCH_1) ==
                          BSP_SWITCH_STATE_PRESSED )
                    {
                        appData.transmitDataBuffer[0] = 0x00;
                    }
                    else
                    {
                        appData.transmitDataBuffer[0] = 0x01;
                    }

                    appData.hidDataTransmitted = false;
                    USB_DEVICE_HID_ReportSend (USB_DEVICE_HID_INDEX_0,
                        &appData.txTransferHandle, appData.transmitDataBuffer, 1);
                    appData.stateUSB = USB_STATE_SCHEDULE_READ;
                }

                break;
        }
    }
    else
    {

        /* Reset the USB Task State Machine */
        appData.stateUSB = USB_STATE_INIT;

    }
}
```

STEP 11: Schedule a New Report Receive Request

To receive a report from the USB host, you need to schedule a report receive request by using the API provided for the USB HID Function Driver.

Before scheduling the request, the reception flag needs to be set to false to check when the request is completed (you set it to true in the report received complete event in Step 9).

After scheduling the request, the USB State Machine state needs to be moved to the waiting for data state.

Inside the schedule read state of the switch statement of the USB_Task() function, add the code to set the reception flag to false, then schedule a new report receive request and finally set the USB State Machine state member of appData structure to the state that waits for data from the USB host (e.g. USB_STATE_WAITING_FOR_DATA).

```
case USB_STATE_SCHEDULE_READ:

   appData.hidDataReceived = false;
    USB_DEVICE_HID_ReportReceive (USB_DEVICE_HID_INDEX_0,
                    &appData.rxTransferHandle, appData.receiveDataBuffer, 64 );
    appData.stateUSB = USB_STATE_WAITING_FOR_DATA;

    break;
```

STEP 12: Receive, Prepare and Send Reports

When the report is received, the reception flag is set to true; that means there is valid data in the reception buffer. Inside the switch of the USB_Task() function the state is set to USB_STATE_WAITING_FOR_DATA and are checked the next commands that are sent by the Linux USB host driver:

- **0x01**: Toggle the LED1. The state is set to USB_STATE_SCHEDULE_READ.
- **0x02**: Toggle the LED2. The state is set to USB_STATE_SCHEDULE_READ.
- **0x03**: Toggle the LED3. The state is set to USB_STATE_SCHEDULE_READ.
- **0x00**: The USB device gets the Pushbutton state. The state is set to USB_STATE_SEND_REPORT. The HID device replies with a report to the host, where the first byte is the status of the S1 button ("0x00" pressed, "0x01" not pressed).

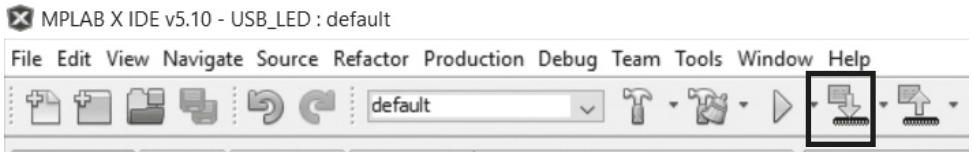
```
case USB_STATE_WAITING_FOR_DATA:

   if( appData.hidDataReceived )
   {
          if (appData.receiveDataBuffer[0]==0x01)
          {
                 BSP_LED_1Toggle();
                 appData.stateUSB = USB_STATE_SCHEDULE_READ;
           }
           else if (appData.receiveDataBuffer[0]==0x02)
           {
```

```
                BSP_LED_2Toggle();
                appData.stateUSB = USB_STATE_SCHEDULE_READ;
            }
            else if (appData.receiveDataBuffer[0]==0x03)
            {
                BSP_LED_3Toggle();
                appData.stateUSB = USB_STATE_SCHEDULE_READ;
            }
            else if (appData.receiveDataBuffer[0]==0x00)
            {
                appData.stateUSB = USB_STATE_SEND_REPORT;
            }
            else
            {
                appData.stateUSB = USB_STATE_SCHEDULE_READ;
            }
        }

        break;

    case USB_STATE_SEND_REPORT:

        if(appData.hidDataTransmitted)
        {
            if( BSP_SwitchStateGet(BSP_SWITCH_1) == BSP_SWITCH_STATE_PRESSED )
            {
                appData.transmitDataBuffer[0] = 0x00;
            }
            else
            {
                appData.transmitDataBuffer[0] = 0x01;
            }

            appData.hidDataTransmitted = false;
            USB_DEVICE_HID_ReportSend (USB_DEVICE_HID_INDEX_0,
                    &appData.txTransferHandle, appData.transmitDataBuffer, 1);
            appData.stateUSB = USB_STATE_SCHEDULE_READ;
        }
```

STEP 13: Program The Application

Power the PIC32MX470 Curiosity Development Board from a Host PC through a Type-A male to mini-B USB cable connected to Mini-B port (J3). Ensure that a jumper is placed in J8 header (between 4 & 3) to select supply from debug USB connector.

Build the code and program the device by clicking on the program button as shown below.



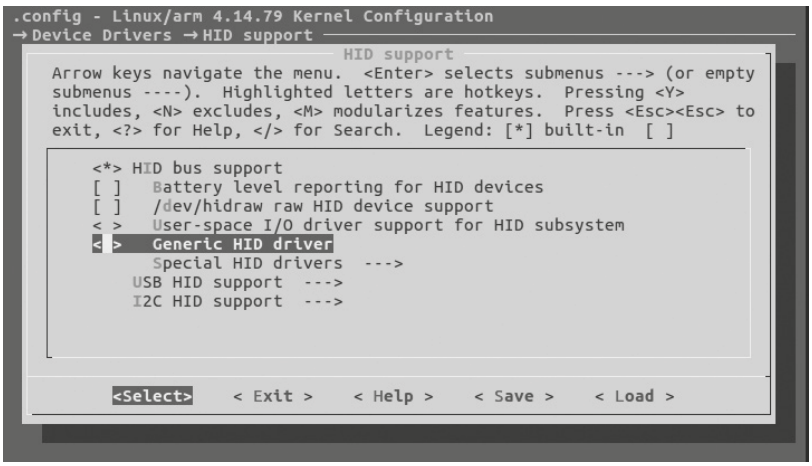
LAB 13.2: "USB LED" Module

In the previous lab, you developed the firmware for a fully functional USB HID device that is able to send and receive data by using HID reports. Now, you are going to develop a Linux USB host driver to control that USB device. The driver will send USB commands to toggle LED1, LED2 and LED3 of the PIC32MX470 Curiosity Development Board; it will receive the command from the Linux user space through a sysfs entry and then retransmit it to the PIC32MX HID device. The command values can be 0x01, 0x02, or 0x03. The HID device must toggle LED1 when it receives 0x01, LED2 when it receives 0x02 and LED3 when it receives 0x03 in the report . You will use the SAMA5D27-SOM1-EK1 evaluation kit to implement this driver. The user guide and design files for the board can be found at https://www.microchip.com/developmenttools/ProductDetails/atsama5d27-som1-ek1

Read the "Yocto Project Setup and Image Building", "Connect and Set Up Hardware", "Working Outside of Yocto", and "Building the Linux Kernel" sections of the "Appendix" of this book to set up the software and hardware for the development of the labs.

Before developing this first driver, you have to remove the Generic HID driver from the kernel. Open the menuconfig window. Navigate from the **main menu -> Device Drivers -> HID bus support -> Generic HID drivers**. Hit <spacebar> until you see a < > appear next to the new configuration. Hit <Exit> until you exit the menuconfig GUI and remember to save the new configuration.

```
~/linux-at91$: make menuconfig ARCH=arm
```



Go to the "Building the Linux Kernel" section of the "Appendix" of this book to check the instructions to compile and load the **sama5d27_som1_ek.itb** FIT image to the target processor.

Connect the USB Micro-B port (J12) of the PIC32MX470 Curiosity Development Board to the J19 USB-B type C connector of the SAMA5D27-SOM1-EK1 board through an USB 2.0 Type-C male to micro-B male cable.

LAB 13.2 Code Description of the "USB LED" Module

The main code sections of the driver will now be described:

1. Include the function headers:

```
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/usb.h>
```

2. Create the ID table to support hotplugging. The Vendor ID and Product ID values have to match with the ones used in the PIC32MX USB HID device.

```
#define USBLED_VENDOR_ID     0x04D8
#define USBLED_PRODUCT_ID    0x003F

/* table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
      { USB_DEVICE(USBLED_VENDOR_ID, USBLED_PRODUCT_ID) },
      { }
};
MODULE_DEVICE_TABLE(usb, id_table);
```

3. Create a private structure that will store the driver´s data.

```
struct usb_led {
      struct usb_device *udev;
      u8 led_number;
};
```

4. See below an extract of the probe() routine with the main lines of code to set up the driver commented.

```
static int led_probe(struct usb_interface *interface,
                     const struct usb_device_id *id)
{
      /* Get the usb_device structure from the usb_interface one */
      struct usb_device *udev = interface_to_usbdev(interface);
      struct usb_led *dev = NULL;
      int retval = -ENOMEM;

      dev_info(&interface->dev, "led_probe() function is called.\n");
```

```
        /* Allocate our private data structure */
        dev = kzalloc(sizeof(struct usb_led), GFP_KERNEL);

        /* store the usb device in our data structure */
        dev->udev = usb_get_dev(udev);

        /* Attach the USB device data to the USB interface */
        usb_set_intfdata(interface, dev);

        /* create a led sysfs entry to interact with the user space */
        device_create_file(&interface->dev, &dev_attr_led);

        return 0;

}
```

5. Write the led_store() function. Every time your user space application writes to the led sysfs entry (/sys/bus/usb/devices/2-2:1.0/led) under the USB device, the driver´s led_store() function is called. The usb_led structure associated to the USB device is recovered by using the usb_get_intfdata() function. The command written to the led sysfs entry is stored in the val variable. Finally, you will send the command value via USB by using the usb_bulk_msg() function.

   The kernel provides two usb_bulk_msg() and usb_control_msg() helper functions that make it possible to transfer simple bulk and control messages without having to create an urb structure, initialize it, submit it and wait for its completion handler. These functions are synchronous and will make your code sleep. You must not call them from interrupt context or with a spinlock held.

   ```
   int usb_bulk_msg(struct usb_device * usb_dev, unsigned int pipe, void * data,
   int len, int * actual_length, int timeout);
   ```

   See below a short description for the usb_bulk_msg() parameters:

   - **usb_dev**: pointer to the usb device to send the message to
   - **pipe**: endpoint "pipe" to send the message to
   - **data**: pointer to the data to send
   - **len**: length in bytes of the data to send
   - **actual_length**: pointer to a location to put the actual length transferred in bytes
   - **timeout**: time in msecs to wait for the message to complete before timing out

See below an extract of the led_store() routine:

```
static ssize_t led_store(struct device *dev, struct device_attribute *attr,
                      const char *buf, size_t count)
{
      struct usb_interface *intf = to_usb_interface(dev);
      struct usb_led *led = usb_get_intfdata(intf);
      u8 val;

      /* transform char array to u8 value */
      kstrtou8(buf, 10, &val);

      led->led_number = val;

      /* Toggle led */
      usb_bulk_msg(led->udev, usb_sndctrlpipe(led->udev, 1),
                   &led->led_number,
                   1,
                   NULL,
                   0);

      return count;
}
static DEVICE_ATTR_RW(led);
```

6. Add a struct usb_driver structure that will be registered to the USB core:

```
static struct usb_driver led_driver = {
      .name =        "usbled",
      .probe =       led_probe,
      .disconnect =  led_disconnect,
      .id_table =    id_table,
};
```

7. Register your driver with the USB bus:

```
module_usb_driver(led_driver);
```

8. Build the module and load it to the target processor:

```
/* driver´s source code and Makefile are stored in the linux_usb_drivers folder
*/
~/linux_usb_drivers$ . /opt/poky-atmel/2.5.1/environment-setup-cortexa5hf-neon-
poky-linux-gnueabi

/* Boot the target. Mount the SD */
root@sama5d27-som1-ek-sd:~# mount /dev/mmcblk0p1 /mnt

/* compile and send the driver to the target */
~/linux_usb_drivers$ make
~/linux_usb_drivers$ make deploy
```

```
/* Umount the SD and reboot */
root@sama5d27-som1-ek-sd:~# umount /mnt
root@sama5d27-som1-ek-sd:~# reboot
```

See in the next **Listing 13-1** the "USB LED" driver source code (usb_led.c) for the SAMA5D27-SOM1 device.

Note: The source code of the driver and the Makefile can be downloaded from the GitHub repository of this book.

Listing 13-1: usb_led.c

```
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/usb.h>

#define USBLED_VENDOR_ID  0x04D8
#define USBLED_PRODUCT_ID 0x003F

/* table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
   { USB_DEVICE(USBLED_VENDOR_ID, USBLED_PRODUCT_ID) },
   { }
};
MODULE_DEVICE_TABLE(usb, id_table);

struct usb_led {
   struct usb_device *udev;
   u8 led_number;
};

static ssize_t led_show(struct device *dev, struct device_attribute *attr,
                     char *buf)
{
   struct usb_interface *intf = to_usb_interface(dev);
   struct usb_led *led = usb_get_intfdata(intf);

   return sprintf(buf, "%d\n", led->led_number);
}

static ssize_t led_store(struct device *dev, struct device_attribute *attr,
                   const char *buf, size_t count)
{
   struct usb_interface *intf = to_usb_interface(dev);
   struct usb_led *led = usb_get_intfdata(intf);
   u8 val;
```

```
    int error, retval;
    dev_info(&intf->dev, "led_store() function is called.\n");

    /* transform char array to u8 value */
    error = kstrtou8(buf, 10, &val);
    if (error)
            return error;

    led->led_number = val;

    if (val == 1 || val == 2 || val == 3)
            dev_info(&led->udev->dev, "led = %d\n", led->led_number);
    else {
            dev_info(&led->udev->dev, "unknown led %d\n", led->led_number);
            retval = -EINVAL;
            return retval;
    }

    /* Toggle led */
    retval = usb_bulk_msg(led->udev, usb_sndctrlpipe(led->udev, 1),
                          &led->led_number,
                          1,
                          NULL,
                          0);
    if (retval) {
            retval = -EFAULT;
            return retval;
    }
    return count;
}
static DEVICE_ATTR_RW(led);

static int led_probe(struct usb_interface *interface,
                const struct usb_device_id *id)
{
    struct usb_device *udev = interface_to_usbdev(interface);
    struct usb_led *dev = NULL;
    int retval = -ENOMEM;

    dev_info(&interface->dev, "led_probe() function is called.\n");

    dev = kzalloc(sizeof(struct usb_led), GFP_KERNEL);
    if (!dev) {
            dev_err(&interface->dev, "out of memory\n");
            retval = -ENOMEM;
            goto error;
    }
```

```
    dev->udev = usb_get_dev(udev);

    usb_set_intfdata(interface, dev);

    retval = device_create_file(&interface->dev, &dev_attr_led);
    if (retval)
            goto error_create_file;

    return 0;

error_create_file:
    usb_put_dev(udev);
    usb_set_intfdata(interface, NULL);
error:
    kfree(dev);
    return retval;
}

static void led_disconnect(struct usb_interface *interface)
{
    struct usb_led *dev;

    dev = usb_get_intfdata(interface);

    device_remove_file(&interface->dev, &dev_attr_led);
    usb_set_intfdata(interface, NULL);
    usb_put_dev(dev->udev);
    kfree(dev);

    dev_info(&interface->dev, "USB LED now disconnected\n");
}

static struct usb_driver led_driver = {
    .name =        "usbled",
    .probe =       led_probe,
    .disconnect =  led_disconnect,
    .id_table =    id_table,
};

module_usb_driver(led_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a synchronous led usb controlled module");
```

usb_led.ko Demonstration

```
/*
 * Connect the PIC32MX470 Curiosity Development Board USB Micro-B port (J12) to
 * the SAMA5D27-SOM1-EK1 J19 USB-B type C connector through a USB 2.0 Type-C male
 * to micro-B male cable. Power the SAMA5D27 board to boot the processor. Keep the
 * PIC32MX470 board powered off
 */
root@sama5d27-som1-ek-sd:~# insmod usb_led.ko /* load the module */
usb_led: loading out-of-tree module taints kernel.
usbcore: registered new interface driver usbled

/* power now the PIC32MX Curiosity board */
root@sama5d27-som1-ek-sd:~# usb 2-2: new full-speed USB device number 5 using at
91_ohci
usb 2-2: New USB device found, idVendor=04d8, idProduct=003f
usb 2-2: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 2-2: Product: LED_USB HID Demo
usb 2-2: Manufacturer: Microchip Technology Inc.
usbled 2-2:1.0: led_probe() function is called.

/* check the new created USB device */
root@sama5d27-som1-ek-sd:~# cd /sys/bus/usb/devices/2-2:1.0
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0# ls
authorized         bInterfaceProtocol  ep_01     power
bAlternateSetting  bInterfaceSubClass  ep_81     subsystem
bInterfaceClass    bNumEndpoints       led       supports_autosuspend
bInterfaceNumber   driver              modalias  uevent

/* Read the configurations of the USB device */
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0# cat bNumEndpoints
02
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0/ep_01# cat direction
out
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0/ep_81# cat direction
in
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0# cat bAlternateSetting
 0
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0# cat bInterfaceClass
03
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0# cat bNumEndpoints
02

/* Switch on the LED1 of the PIC32MX Curiosity board */
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0# echo 1 > led
usbled 2-2:1.0: led_store() function is called.
usb 2-2: led = 1
```

```
/* Switch on the LED2 of the PIC32MX Curiosity board */
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0# echo 2 > led
usbled 2-2:1.0: led_store() function is called.
usb 2-2: led = 2

/* Switch on the LED3 of the PIC32MX Curiosity board */
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0# echo 3 > led
usbled 2-2:1.0: led_store() function is called.
usb 2-2: led = 3

/* read the led status */
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0# cat led
3

root@sama5d27-som1-ek-sd:~# rmmod usb_led /* remove the module */
usbcore: deregistering interface driver usbled
usbled 2-2:1.0: USB LED now disconnected
```

LAB 13.3: "USB LED and Switch" Module

In this new lab, you will increase the functionality of the previous driver. Besides controlling three LEDs connected to the USB device, the Linux host driver will receive a Pushbutton (S1 switch of the PIC32MX470 Curiosity Development Board) state from the USB HID device. The driver will send a command to the USB device with value 0x00, then the HID device will reply with a report, where the first byte is the status of the S1 button ("0x00" pressed, "0x01" not pressed). In this driver, unlike the previous one, the communication between the host and the device is done asynchronously by using USB Request Blocks (urbs).

LAB 13.3 Code Description of the "USB LED and Switch" Module

The main code sections of the driver will now be described:

1. Include the function headers:

```
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/usb.h>
```

2. Create the ID table to support hotplugging. The Vendor ID and Product ID values have to match with the ones used in the PIC32MX USB HID device.

```
#define USBLED_VENDOR_ID     0x04D8
#define USBLED_PRODUCT_ID    0x003F

/* table of devices that work with this driver */
```

```
static const struct usb_device_id id_table[] = {
      { USB_DEVICE(USBLED_VENDOR_ID, USBLED_PRODUCT_ID) },
      { }
};
MODULE_DEVICE_TABLE(usb, id_table);
```

3. Create a private structure that will store the driver´s data.

```
struct usb_led {
      struct usb_device     *udev;
      struct usb_interface  *intf;
      struct urb            *interrupt_out_urb;
      struct urb            *interrupt_in_urb;
      struct usb_endpoint_descriptor *interrupt_out_endpoint;
      struct usb_endpoint_descriptor *interrupt_in_endpoint;
      u8             irq_data;
      u8             led_number;
      u8             ibuffer;
      int            interrupt_out_interval;
      int            ep_in;
      int            ep_out;
};
```

4. See below an extract of the probe() routine with the main lines of code to configure the driver commented.

```
static int led_probe(struct usb_interface *intf,
                  const struct usb_device_id *id)
{
      struct usb_device *udev = interface_to_usbdev(intf);

      /* Get the current altsetting of the USB interface */
      struct usb_host_interface *altsetting = intf->cur_altsetting;
      struct usb_endpoint_descriptor *endpoint;
      struct usb_led *dev = NULL;
      int ep;
      int ep_in, ep_out;
      int size;

      /*
       * Find the last interrupt out endpoint descriptor
       * to check its number and its size
       * Just for teaching purposes
       */
      usb_find_last_int_out_endpoint(altsetting, &endpoint);

      /* get the endpoint's number */
      ep = usb_endpoint_num(endpoint); /* value from 0 to 15, it is 1 */
      size = usb_endpoint_maxp(endpoint);
```

```
/* Validate endpoint and size */
if (size <= 0) {
       dev_info(&intf->dev, "invalid size (%d)", size);
       return -ENODEV;
}

dev_info(&intf->dev, "endpoint size is (%d)", size);
dev_info(&intf->dev, "endpoint number is (%d)", ep);

/* Get the two addresses (IN and OUT) of the Endpoint 1 */
ep_in = altsetting->endpoint[0].desc.bEndpointAddress;
ep_out = altsetting->endpoint[1].desc.bEndpointAddress;

/* Allocate our private data structure */
dev = kzalloc(sizeof(struct usb_led), GFP_KERNEL);

/* Store values in the data structure */
dev->ep_in = ep_in;
dev->ep_out = ep_out;
dev->udev = usb_get_dev(udev);
dev->intf = intf;

/* allocate the int_out_urb structure */
dev->interrupt_out_urb = usb_alloc_urb(0, GFP_KERNEL);

/* initialize the int_out_urb */
usb_fill_int_urb(dev->interrupt_out_urb,
              dev->udev,
              usb_sndintpipe(dev->udev, ep_out),
              (void *)&dev->irq_data,
              1,
              led_urb_out_callback, dev, 1);

/* allocate the int_in_urb structure */
dev->interrupt_in_urb = usb_alloc_urb(0, GFP_KERNEL);
if (!dev->interrupt_in_urb)
       goto error_out;

/* initialize the int_in_urb */
usb_fill_int_urb(dev->interrupt_in_urb,
              dev->udev,
              usb_rcvintpipe(dev->udev, ep_in),
              (void *)&dev->ibuffer,
              1,
              led_urb_in_callback, dev, 1);

/* Attach the device data to the interface */
usb_set_intfdata(intf, dev);
```

```
        /* create the led sysfs entry to interact with the user space */
        device_create_file(&intf->dev, &dev_attr_led);

        /* Submit the interrrupt IN URB */
        usb_submit_urb(dev->interrupt_in_urb, GFP_KERNEL);

        return 0;
}
```

5. Write the led_store() function. Every time your user space application writes to the led sysfs entry (/sys/bus/usb/devices/2-2:1.0/led) under the USB device, the driver´s led_store() function is called. The usb_led structure associated to the USB device is recovered by using the usb_get_intfdata() function. The command written to the led sysfs entry is stored in the irq_data variable. Finally, you will send the command value via USB by using the usb_submit_urb() function.

   See below an extract of the led_store() routine:

```
static ssize_t led_store(struct device *dev, struct device_attribute *attr,
                         const char *buf, size_t count)
{
        struct usb_interface *intf = to_usb_interface(dev);
        struct usb_led *led = usb_get_intfdata(intf);
        u8 val;

        /* transform char array to u8 value */
        kstrtou8(buf, 10, &val);

        led->irq_data = val;

        /* send the data out */
        retval = usb_submit_urb(led->interrupt_out_urb, GFP_KERNEL);

        return count;
}
static DEVICE_ATTR_RW(led);
```

6. Create OUT and IN URB's completion callbacks. The interrupt OUT completion callback merely checks the URB status and returns. The interrupt IN completion callback checks the URB status, then reads the ibuffer to know the status received from the PIC32MX board´s S1 switch, and finally re-submits the interrupt IN URB.

```
static void led_urb_out_callback(struct urb *urb)
{
        struct usb_led *dev;

        dev = urb->context;
```

```
        /* sync/async unlink faults aren't errors */
        if (urb->status) {
                if (!(urb->status == -ENOENT ||
                    urb->status == -ECONNRESET ||
                    urb->status == -ESHUTDOWN))
                       dev_err(&dev->udev->dev,
                               "%s - nonzero write status received: %d\n",
                               __func__, urb->status);
        }
}

static void led_urb_in_callback(struct urb *urb)
{
        int retval;
        struct usb_led *dev;

        dev = urb->context;

        if (urb->status) {
                if (!(urb->status == -ENOENT ||
                    urb->status == -ECONNRESET ||
                    urb->status == -ESHUTDOWN))
                       dev_err(&dev->udev->dev,
                               "%s - nonzero write status received: %d\n",
                               __func__, urb->status);
        }

        if (dev->ibuffer == 0x00)
                pr_info ("switch is ON.\n");
        else if (dev->ibuffer == 0x01)
                pr_info ("switch is OFF.\n");
        else
                pr_info ("bad value received\n");

        usb_submit_urb(dev->interrupt_in_urb, GFP_KERNEL);
}
```

7. Add a struct usb_driver structure that will be registered to the USB core:

```
static struct usb_driver led_driver = {
        .name =         "usbled",
        .probe =        led_probe,
        .disconnect =   led_disconnect,
        .id_table =     id_table,
};
```

8. Register your driver with the USB bus:

```
module_usb_driver(led_driver);
```

9. Build the module and load it to the target processor:

```
/* driver´s source code and Makefile are stored in the linux_usb_drivers folder
*/
~/linux_usb_drivers$ . /opt/poky-atmel/2.5.1/environment-setup-cortexa5hf-neon-
poky-linux-gnueabi

/* Boot the target. Mount the SD */
root@sama5d27-som1-ek-sd:~# mount /dev/mmcblk0p1 /mnt

/* compile and send the driver to the target */
~/linux_usb_drivers$ make
~/linux_usb_drivers$ make deploy

/* Umount the SD and reboot */
root@sama5d27-som1-ek-sd:~# umount /mnt
root@sama5d27-som1-ek-sd:~# reboot
```

See in the next **Listing 13-2** the "USB LED and Switch" driver source code (usb_urb_int_led.c) for the SAMA5D27-SOM1 device.

Note: The source code of the driver and the Makefile can be downloaded from the GitHub repository of this book.

Listing 13-2: usb_urb_int_led.c

```
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/usb.h>

#define USBLED_VENDOR_ID  0x04D8
#define USBLED_PRODUCT_ID 0x003F

static void led_urb_out_callback(struct urb *urb);
static void led_urb_in_callback(struct urb *urb);

/* table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
   { USB_DEVICE(USBLED_VENDOR_ID, USBLED_PRODUCT_ID) },
   { }
};
MODULE_DEVICE_TABLE(usb, id_table);

struct usb_led {
   struct usb_device *udev;
   struct usb_interface *intf;
   struct urb       *interrupt_out_urb;
   struct urb       *interrupt_in_urb;
```

```
    struct usb_endpoint_descriptor *interrupt_out_endpoint;
    struct usb_endpoint_descriptor *interrupt_in_endpoint;
    u8              irq_data;
    u8              led_number;
    u8              ibuffer;
    int             interrupt_out_interval;
    int ep_in;
    int ep_out;
};

static ssize_t led_show(struct device *dev, struct device_attribute *attr,
                    char *buf)
{
    struct usb_interface *intf = to_usb_interface(dev);
    struct usb_led *led = usb_get_intfdata(intf);

    return sprintf(buf, "%d\n", led->led_number);
}

static ssize_t led_store(struct device *dev, struct device_attribute *attr,
                   const char *buf, size_t count)
{
    struct usb_interface *intf = to_usb_interface(dev);
    struct usb_led *led = usb_get_intfdata(intf);
    u8 val;
    int error, retval;

    dev_info(&intf->dev, "led_store() function is called.\n");

    /* transform char array to u8 value */
    error = kstrtou8(buf, 10, &val);
    if (error)
           return error;

    led->led_number = val;
    led->irq_data = val;

    if (val == 0)
           dev_info(&led->udev->dev, "read status\n");
    else if (val == 1 || val == 2 || val == 3)
           dev_info(&led->udev->dev, "led = %d\n", led->led_number);
     else {
           dev_info(&led->udev->dev, "unknown value %d\n", val);
           retval = -EINVAL;
           return retval;
     }
```

```
    /* send the data out */
    retval = usb_submit_urb(led->interrupt_out_urb, GFP_KERNEL);
    if (retval) {
            dev_err(&led->udev->dev,
                    "Couldn't submit interrupt_out_urb %d\n", retval);
            return retval;
    }

    return count;
}
static DEVICE_ATTR_RW(led);

static void led_urb_out_callback(struct urb *urb)
{
    struct usb_led *dev;

    dev = urb->context;

    dev_info(&dev->udev->dev, "led_urb_out_callback() function is called.\n");

    /* sync/async unlink faults aren't errors */
    if (urb->status) {
            if (!(urb->status == -ENOENT ||
                urb->status == -ECONNRESET ||
                urb->status == -ESHUTDOWN))
                    dev_err(&dev->udev->dev,
                            "%s - nonzero write status received: %d\n",
                            __func__, urb->status);
    }
}

static void led_urb_in_callback(struct urb *urb)
{
    int retval;
    struct usb_led *dev;

    dev = urb->context;

    dev_info(&dev->udev->dev, "led_urb_in_callback() function is called.\n");

    if (urb->status) {
            if (!(urb->status == -ENOENT ||
                urb->status == -ECONNRESET ||
                urb->status == -ESHUTDOWN))
                    dev_err(&dev->udev->dev,
                            "%s - nonzero write status received: %d\n",
                            __func__, urb->status);
    }
```

```
    if (dev->ibuffer == 0x00)
            pr_info ("switch is ON.\n");
    else if (dev->ibuffer == 0x01)
            pr_info ("switch is OFF.\n");
    else
            pr_info ("bad value received\n");

    retval = usb_submit_urb(dev->interrupt_in_urb, GFP_KERNEL);
    if (retval)
            dev_err(&dev->udev->dev,
                    "Couldn't submit interrupt_in_urb %d\n", retval);
}

static int led_probe(struct usb_interface *intf,
                const struct usb_device_id *id)
{
    struct usb_device *udev = interface_to_usbdev(intf);
    struct usb_host_interface *altsetting = intf->cur_altsetting;
    struct usb_endpoint_descriptor *endpoint;
    struct usb_led *dev = NULL;
    int ep;
    int ep_in, ep_out;
    int retval, size, res;

    dev_info(&intf->dev, "led_probe() function is called.\n");

    res = usb_find_last_int_out_endpoint(altsetting, &endpoint);
    if (res) {
            dev_info(&intf->dev, "no endpoint found");
            return res;
    }

    ep = usb_endpoint_num(endpoint); /* value from 0 to 15, it is 1 */
    size = usb_endpoint_maxp(endpoint);

    /* Validate endpoint and size */
    if (size <= 0) {
            dev_info(&intf->dev, "invalid size (%d)", size);
            return -ENODEV;
    }

    dev_info(&intf->dev, "endpoint size is (%d)", size);
    dev_info(&intf->dev, "endpoint number is (%d)", ep);

    ep_in = altsetting->endpoint[0].desc.bEndpointAddress;
    ep_out = altsetting->endpoint[1].desc.bEndpointAddress;

    dev_info(&intf->dev, "endpoint in address is (%d)", ep_in);
```

```
dev_info(&intf->dev, "endpoint out address is (%d)", ep_out);

dev = kzalloc(sizeof(struct usb_led), GFP_KERNEL);

if (!dev)
        return -ENOMEM;

dev->ep_in = ep_in;
dev->ep_out = ep_out;

dev->udev = usb_get_dev(udev);

dev->intf = intf;

/* allocate int_out_urb structure */
dev->interrupt_out_urb = usb_alloc_urb(0, GFP_KERNEL);
if (!dev->interrupt_out_urb)
        goto error_out;

/* initialize int_out_urb */
usb_fill_int_urb(dev->interrupt_out_urb,
                dev->udev,
                usb_sndintpipe(dev->udev, ep_out),
                (void *)&dev->irq_data,
                1,
                led_urb_out_callback, dev, 1);

/* allocate int_in_urb structure */
dev->interrupt_in_urb = usb_alloc_urb(0, GFP_KERNEL);
if (!dev->interrupt_in_urb)
        goto error_out;

/* initialize int_in_urb */
usb_fill_int_urb(dev->interrupt_in_urb,
                dev->udev,
                usb_rcvintpipe(dev->udev, ep_in),
                (void *)&dev->ibuffer,
                1,
                led_urb_in_callback, dev, 1);

usb_set_intfdata(intf, dev);

retval = device_create_file(&intf->dev, &dev_attr_led);
if (retval)
        goto error_create_file;

retval = usb_submit_urb(dev->interrupt_in_urb, GFP_KERNEL);
if (retval) {
```

```
            dev_err(&dev->udev->dev,
                    "Couldn't submit interrupt_in_urb %d\n", retval);
            device_remove_file(&intf->dev, &dev_attr_led);
            goto error_create_file;
    }

    dev_info(&dev->udev->dev,"int_in_urb submitted\n");

    return 0;

error_create_file:
    usb_free_urb(dev->interrupt_out_urb);
    usb_free_urb(dev->interrupt_in_urb);
    usb_put_dev(udev);
    usb_set_intfdata(intf, NULL);

error_out:
    kfree(dev);
    return retval;
}

static void led_disconnect(struct usb_interface *interface)
{
    struct usb_led *dev;

    dev = usb_get_intfdata(interface);

    device_remove_file(&interface->dev, &dev_attr_led);
    usb_free_urb(dev->interrupt_out_urb);
    usb_free_urb(dev->interrupt_in_urb);
    usb_set_intfdata(interface, NULL);
    usb_put_dev(dev->udev);
    kfree(dev);

    dev_info(&interface->dev, "USB LED now disconnected\n");
}

static struct usb_driver led_driver = {
    .name =        "usbled",
    .probe =       led_probe,
    .disconnect =  led_disconnect,
    .id_table =    id_table,
};

module_usb_driver(led_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
```

```
MODULE_DESCRIPTION("This is a led/switch usb controlled module with irq in/out
endpoints");
```

usb_urb_int_led.ko Demonstration

```
/*
 * Connect the PIC32MX470 Curiosity Development Board USB Micro-B port (J12) to
 * the SAMA5D27-SOM1-EK1 J19 USB-B type C connector through a USB 2.0 Type-C male
 * to micro-B male cable. Power the SAMA5D27 board to boot the processor. Keep the
 * PIC32MX470 board powered off
 */
root@sama5d27-som1-ek-sd:~# insmod usb_urb_int_led.ko /* load the module */
usbcore: registered new interface driver usbled

/* power now the PIC32MX Curiosity board */
root@sama5d27-som1-ek-sd:~# usb 2-2: new full-speed USB device number 6 using at
91_ohci
usb 2-2: New USB device found, idVendor=04d8, idProduct=003f
usb 2-2: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 2-2: Product: LED_USB HID Demo
usb 2-2: Manufacturer: Microchip Technology Inc.
usbled 2-2:1.0: led_probe() function is called.
usbled 2-2:1.0: endpoint size is (64)
usbled 2-2:1.0: endpoint number is (1)
usbled 2-2:1.0: endpoint in address is (129)
usbled 2-2:1.0: endpoint out address is (1)
usb 2-2: int_in_urb submitted

/* Go to the new created USB device */
root@sama5d27-som1-ek-sd:~# cd /sys/bus/usb/devices/2-2:1.0

/* Switch on the LED1 of the PIC32MX Curiosity board */
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0# echo 1 > led
usbled 2-2:1.0: led_store() function is called.
usb 2-2: led = 1
usb 2-2: led_urb_out_callback() function is called.

/* Switch on the LED2 of the PIC32MX Curiosity board */
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0# echo 2 > led
usbled 2-2:1.0: led_store() function is called.
usb 2-2: led = 2
usb 2-2: led_urb_out_callback() function is called.

/* Switch on the LED3 of the PIC32MX Curiosity board */
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0# echo 3 > led
usbled 2-2:1.0: led_store() function is called.
```

```
usb 2-2: led = 3
usb 2-2: led_urb_out_callback() function is called.

/* Keep pressed the S1 switch of PIC32MX Curiosity board and get SW status*/
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0# echo 0 > led
usbled 2-2:1.0: led_store() function is called.
usb 2-2: read status
usb 2-2: led_urb_out_callback() function is called.
usb 2-2: led_urb_in_callback() function is called.
switch is ON.

/* Release the S1 switch of PIC32MX Curiosity board and get SW status */
root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0# echo 0 > led
usbled 2-2:1.0: led_store() function is called.
usb 2-2: read status
usb 2-2: led_urb_out_callback() function is called.
usb 2-2: led_urb_in_callback() function is called.
switch is OFF.

root@sama5d27-som1-ek-sd:~# rmmod usb_urb_int_led.ko /* remove the module */
usbcore: deregistering interface driver usbled
usb 2-2: led_urb_in_callback() function is called.
switch is OFF.
usb 2-2: Couldn't submit interrupt_in_urb -1
usbled 2-2:1.0: USB LED now disconnected
```

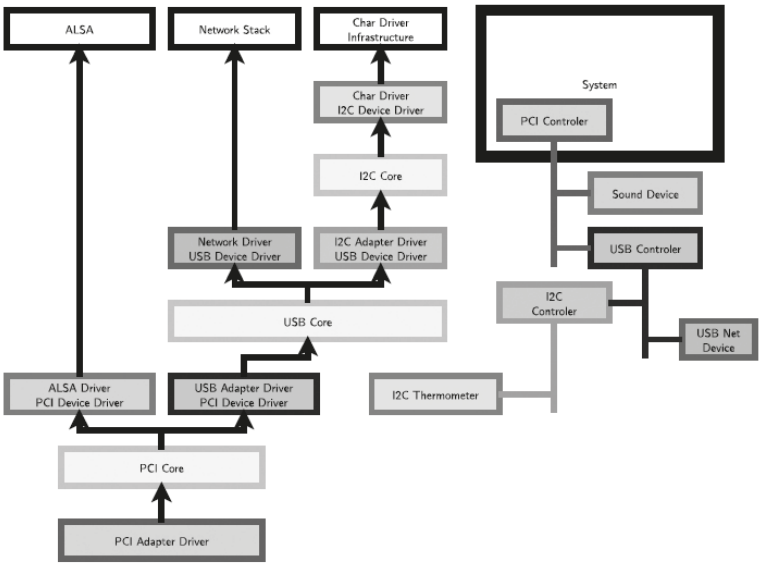
LAB 13.4: "I2C to USB Multidisplay LED" Module

In the lab 6.2 of this book, you implemented a driver to control the Analog Devices LTC3206 I2C Multidisplay LED controller (http://www.analog.com/en/products/power-management/led-driver-ic/inductorless-charge-pump-led-drivers/ltc3206.html). In that lab 6.2, you controlled the LTC3206 device by using an I2C Linux driver. In this lab 13.4, you will write a Linux USB driver that is controlled from the user space by using the I2C Tools for Linux; to perform this task you will have to create a new I2C adapter within your created USB driver.

The driver model is recursive. In the following image, you can see all the needed drivers to control an I2C device through a PCI board that integrates an USB to I2C converter. These are the main steps to create this recursive driver model:

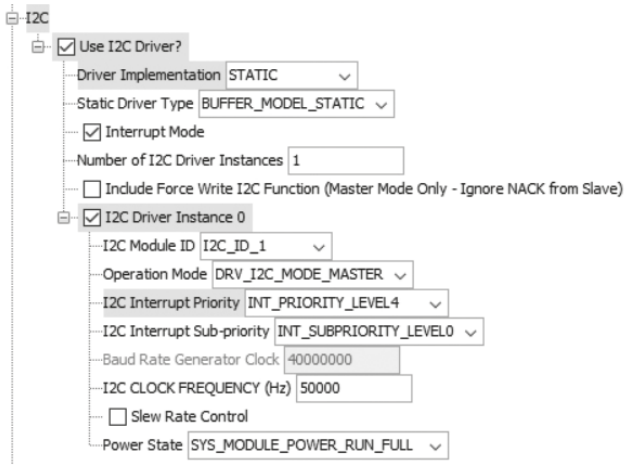
- First, you have to develop a PCI device driver that will create an USB adapter (the PCI device driver is the parent of the USB adapter driver).
- Second, you have to develop an USB device driver that will send USB data to the USB adapter driver through the USB Core; this USB device driver will also create an I2C adapter driver (the USB device driver is the parent of the I2C adapter driver).

- Finally, you will create an I2C device driver that will send data to the I2C adapter driver through the I2C Core and will create a struct file_operations structure to define driver´s functions which are called when the Linux user space reads, and writes to character devices.

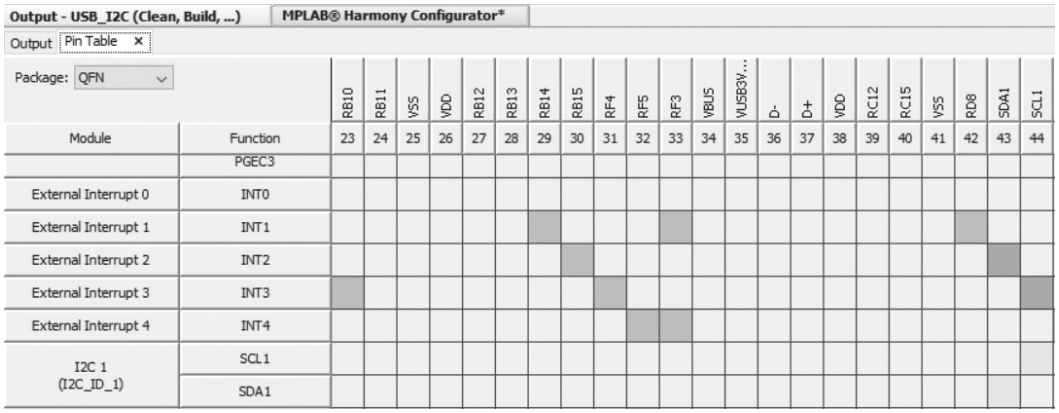


This recursive model will be simplified in the driver of lab 13.4, where you are only going to execute the second step of the three previously mentioned. In this driver, the communication between the host and the device is done asynchronously by using an interrupt OUT URB.

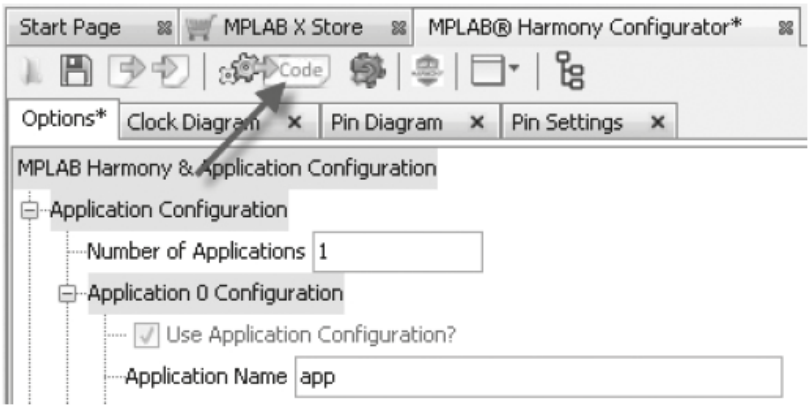
Before developing the Linux driver, you must first add new Harmony configurations to the previous project ones. You must select the I2C Drivers option inside the Harmony Framework Configuration:



In the Pin Table of the MPLAB Harmony Configurator activate the SCL1 and SDA1 pins of the I2C1 controller:



Generate the code, save the modified configuration, and generate the project:



Now, you have to modify the generated app.c code. Go to the USB_STATE_WAITING_FOR_DATA case inside the USB_Task() function. Basically, it is waiting for I2C data, which has been encapsulated inside an USB interrupt OUT URB; once the PIC32MX USB device receives the information, it forwards it via I2C to the LTC3206 device connected to the MikroBus 1 of the PIC32MX470 Curiosity Development Board.

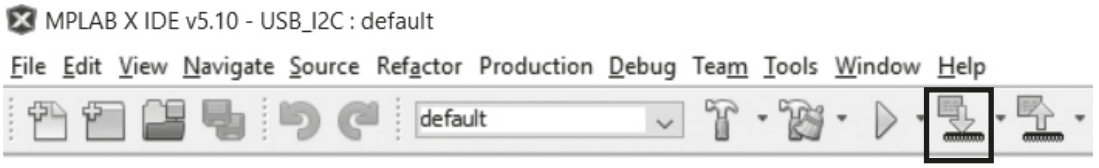
```
static void USB_Task (void)
{
    if(appData.usbDeviceIsConfigured)
    {
```

```
    switch (appData.stateUSB)
    {
        case USB_STATE_INIT:

            appData.hidDataTransmitted = true;
            appData.txTransferHandle = USB_DEVICE_HID_TRANSFER_HANDLE_INVALID;
            appData.rxTransferHandle = USB_DEVICE_HID_TRANSFER_HANDLE_INVALID;
            appData.stateUSB = USB_STATE_SCHEDULE_READ;

            break;

        case USB_STATE_SCHEDULE_READ:

            appData.hidDataReceived = false;

            /* receive from Host (OUT endpoint). It is a write
             command to the LTC3206 device */
            USB_DEVICE_HID_ReportReceive (USB_DEVICE_HID_INDEX_0,
                &appData.rxTransferHandle, appData.receiveDataBuffer, 64 );
            appData.stateUSB = USB_STATE_WAITING_FOR_DATA;

            break;

        case USB_STATE_WAITING_FOR_DATA:

            if( appData.hidDataReceived )
            {

                DRV_I2C_Transmit (appData.drvI2CHandle_Master,
                                  0x36,
                                  &appData.receiveDataBuffer[0],
                                  3,
                                  NULL);

                appData.stateUSB = USB_STATE_SCHEDULE_READ;
            }
            break;
    }
}
else
{

       appData.stateUSB = USB_STATE_INIT;

}
```

You also need to open the I2C driver inside the APP_Tasks() function:

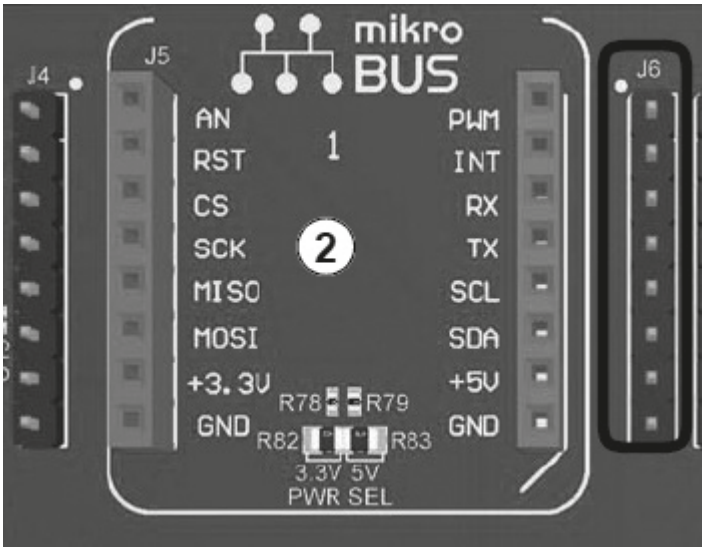
```
/* Application's initial state. */
 case APP_STATE_INIT:
 {
     bool appInitialized = true;

     /* Open the I2C Driver for Slave device */
     appData.drvI2CHandle_Master = DRV_I2C_Open(DRV_I2C_INDEX_0,
                                                DRV_IO_INTENT_WRITE);
     if ( appData.drvI2CHandle_Master == (DRV_HANDLE)NULL )
     {
         appInitialized = false;
     }

     /* Open the device layer */
     if (appData.handleUsbDevice == USB_DEVICE_HANDLE_INVALID)
     {
         appData.handleUsbDevice = USB_DEVICE_Open( USB_DEVICE_INDEX_0,
                                                    DRV_IO_INTENT_READWRITE );
         if(appData.handleUsbDevice != USB_DEVICE_HANDLE_INVALID)
         {
             appInitialized = true;
         }
         else
         {
             appInitialized = false;
         }
     }
```

Now, you must build the code and program the PIC32MX with the new aplication. You can download this new project from the book´s Github.



You will use the LTC3206 **DC749A - Demo Board** (http://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/dc749a.html) to test the driver. You will connect the board to the MikroBUS 1 connector of the Curiosity PIC32MX470 Development Board. Connect the MikroBUS 1 SDA pin to the pin 7 (SDA) of the DC749A J1 connector and the MikroBUS 1 SCL pin to the pin 4 (SCL) of the DC749A J1 connector. Connect the MikroBUS 1 3.3V pin, the DC749A J20 DVCC pin and the DC749A pin 6 (ENRGB/S) to the DC749A Vin J2 pin. Do not forget to connect GND between the two boards.

Note: For the Curiosity PIC32MX470 Development Board, verify that the value of the series resistors mounted on the SCL and SDA lines of the mikroBUS 1 socket J5 are set to zero Ohms. If not, replace them with zero Ohm resistors. You can also take the SDA and SCL signals from the J6 connector if you do not want to replace the resistors.



LAB 13.4 Code Description of the "I2C to USB Multidisplay LED" Module

The main code sections of the driver will now be described:

1. Include the function headers:

```
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/usb.h>
#include <linux/i2c.h>
```

2. Create the ID table to support hotplugging. The Vendor ID and Product ID values have to match with the ones used in the PIC32MX USB HID device.

```
#define USBLED_VENDOR_ID     0x04D8
#define USBLED_PRODUCT_ID    0x003F

/* table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
      { USB_DEVICE(USBLED_VENDOR_ID, USBLED_PRODUCT_ID) },
      { }
};
MODULE_DEVICE_TABLE(usb, id_table);
```

3. Create a private structure that will store the driver´s data.

```
struct i2c_ltc3206 {
      u8 obuffer[LTC3206_OUTBUF_LEN];     /* USB write buffer */
      /* I2C/SMBus data buffer */
      u8 user_data_buffer[LTC3206_I2C_DATA_LEN]; /* LEN is 3 bytes */
      int ep_out;                  /* out endpoint */
      struct usb_device *usb_dev;  /* the usb device for this device */
      struct usb_interface *interface;/* the interface for this device */
      struct i2c_adapter adapter;  /* i2c related things */
      /* wq to wait for an ongoing write */
      wait_queue_head_t usb_urb_completion_wait;
      bool ongoing_usb_ll_op;             /* all is in progress */
      struct urb *interrupt_out_urb;      /* interrupt out URB */
};
```

4. See below an extract of the probe() routine with the main lines of code to configure the driver commented.

```
static int ltc3206_probe(struct usb_interface *interface,
                         const struct usb_device_id *id)
{
      /* Get the current altsetting of the USB interface */
      struct usb_host_interface *hostif = interface->cur_altsetting;
      struct i2c_ltc3206 *dev; /* the data structure */
```

```
        /* allocate data memory for our USB device and initialize it */
        kzalloc(sizeof(*dev), GFP_KERNEL);

        /* get interrupt ep_out address */
        dev->ep_out = hostif->endpoint[1].desc.bEndpointAddress;

        dev->usb_dev = usb_get_dev(interface_to_usbdev(interface));
        dev->interface = interface;

        /* declare dynamically a wait queue */
        init_waitqueue_head(&dev->usb_urb_completion_wait);

        /* save our data pointer in this USB interface device */
        usb_set_intfdata(interface, dev);

        /* setup I2C adapter description */
        dev->adapter.owner = THIS_MODULE;
        dev->adapter.class = I2C_CLASS_HWMON;
        dev->adapter.algo = &ltc3206_usb_algorithm;
        i2c_set_adapdata(&dev->adapter, dev);

        /* Attach the I2C adapter to the USB interface */
        dev->adapter.dev.parent = &dev->interface->dev;

        /* initialize the I2C device */
        ltc3206_init(dev);

        /* and finally attach the adapter to the I2C layer */
        i2c_add_adapter(&dev->adapter);

        return 0;
}
```

5. Write the ltc3206_init() function. Inside this function, you will allocate and initialize the interrupt OUT URB which is used for the communication between the host and the device. See below an extract of the ltc3206_init() routine:

```
static int ltc3206_init(struct i2c_ltc3206 *dev)
{
        /* allocate int_out_urb structure */
        interrupt_out_urb = usb_alloc_urb(0, GFP_KERNEL);

        /* initialize int_out_urb structure */
        usb_fill_int_urb(dev->interrupt_out_urb, dev->usb_dev,
                         usb_sndintpipe(dev->usb_dev,
                                        dev->ep_out),
```

```
                                (void *)&dev->obuffer, LTC3206_OUTBUF_LEN,
                                ltc3206_usb_cmpl_cbk, dev,
                                1);

        return 0;
}
```

6. Create a struct i2c_algorithm that represents the I2C transfer method. You will initialize two variables inside this structure:

   - **master_xfer**: Issues a set of i2c transactions to the given I2C adapter defined by the msgs array, with num messages available to transfer via the adapter specified by adap.
   - **functionality**: Returns the flags that this algorithm/adapter pair supports from the I2C_FUNC_* flags.

```
static const struct i2c_algorithm ltc3206_usb_algorithm = {
        .master_xfer = ltc3206_usb_i2c_xfer,
        .functionality = ltc3206_usb_func,
};
```

7. Write the ltc3206_usb_i2c_xfer() function. This function will be called each time you write to the I2C adapter from the Linux user space. This function will call ltc32016_i2c_write() which stores the I2C data received from the Linux user space in the obuffer[] char array, then ltc32016_i2c_write() will call ltc3206_ll_cmd() which submits the interrupt OUT URB to the USB device and waits for the URB´s completion.

```
static int ltc3206_usb_i2c_xfer(struct i2c_adapter *adap,
                struct i2c_msg *msgs, int num)
{
        /* get the private data structure */
        struct i2c_ltc3206 *dev = i2c_get_adapdata(adap);
        struct i2c_msg *pmsg;
        int ret, count;

        pr_info("number of i2c msgs is = %d\n", num);

        for (count = 0; count < num; count++) {
                pmsg = &msgs[count];
                ret = ltc3206_i2c_write(dev, pmsg);
                if (ret < 0)
                        goto abort;
        }

        /* if all the messages were transferred ok, return "num" */
        ret = num;
abort:
```

```
        return ret;
}

static int ltc3206_i2c_write(struct i2c_ltc3206 *dev,
                                    struct i2c_msg *pmsg)
{
        u8 ucXferLen;
        int rv;
        u8 *pSrc, *pDst;

        /* I2C write lenght */
        ucXferLen = (u8)pmsg->len;

        pSrc = &pmsg->buf[0];
        pDst = &dev->obuffer[0];
        memcpy(pDst, pSrc, ucXferLen);

        pr_info("oubuffer[0] = %d\n", dev->obuffer[0]);
        pr_info("oubuffer[1] = %d\n", dev->obuffer[1]);
        pr_info("oubuffer[2] = %d\n", dev->obuffer[2]);

        rv = ltc3206_ll_cmd(dev);
        if (rv < 0)
                return -EFAULT;

        return 0;
}
static int ltc3206_ll_cmd(struct i2c_ltc3206 *dev)
{
        int rv;

        /*
         * tell everybody to leave the URB alone
         * we are going to write to the LTC3206
         */
        dev->ongoing_usb_ll_op = 1; /* doing USB communication */

        /* submit the interrupt out ep packet */
        if (usb_submit_urb(dev->interrupt_out_urb, GFP_KERNEL)) {
                dev_err(&dev->interface->dev,
                                "ltc3206(ll): usb_submit_urb intr out failed\n");
                dev->ongoing_usb_ll_op = 0;
                return -EIO;
        }

        /* wait for its completion, the USB URB callback will signal it */
        rv = wait_event_interruptible(dev->usb_urb_completion_wait,
```

```
                (!dev->ongoing_usb_ll_op));
        if (rv < 0) {
                dev_err(&dev->interface->dev, "ltc3206(ll): wait
                        interrupted\n");
                goto ll_exit_clear_flag;
        }

        return 0;

ll_exit_clear_flag:
        dev->ongoing_usb_ll_op = 0;
        return rv;
}
```

8. Create the interrupt OUT URB's completion callback. The completion callback checks the URB status and re-submits the URB if there was an error status. If the transmission was successful, the callback wakes up the sleeping process and returns.

```
static void ltc3206_usb_cmpl_cbk(struct urb *urb)
{
        struct i2c_ltc3206 *dev = urb->context;
        int status = urb->status;
        int retval;

        switch (status) {
        case 0:                 /* success */
                break;
        case -ECONNRESET:       /* unlink */
        case -ENOENT:
        case -ESHUTDOWN:
                return;
        /* -EPIPE:  should clear the halt */
        default:                /* error */
                goto resubmit;
        }

        /*
         * wake up the waiting function
         * modify the flag indicating the ll status
         */
        dev->ongoing_usb_ll_op = 0; /* communication is OK */
        wake_up_interruptible(&dev->usb_urb_completion_wait);
        return;

resubmit:
        retval = usb_submit_urb(urb, GFP_ATOMIC);
        if (retval) {
```

```
                dev_err(&dev->interface->dev,
                        "ltc3206(irq): can't resubmit intrerrupt urb, retval %d\n",
                        retval);
        }
}
```

9. Add a struct usb_driver structure that will be registered to the USB core:

```
static struct usb_driver ltc3206_driver = {
      .name = DRIVER_NAME,
      .probe = ltc3206_probe,
      .disconnect = ltc3206_disconnect,
      .id_table = ltc3206_table,
};
```

10. Register your driver with the USB bus:

```
module_usb_driver(ltc3206_driver);
```

11. Build the module and load it to the target processor:

```
/* driver´s source code and Makefile are stored in the linux_usb_drivers folder
*/
~/linux_usb_drivers$ . /opt/poky-atmel/2.5.1/environment-setup-cortexa5hf-neon-
poky-linux-gnueabi

/* Boot the target. Mount the SD */
root@sama5d27-som1-ek-sd:~# mount /dev/mmcblk0p1 /mnt

/* compile and send the driver to the target */
~/linux_usb_drivers$ make
~/linux_usb_drivers$ make deploy

/* Umount the SD and reboot */
root@sama5d27-som1-ek-sd:~# umount /mnt
root@sama5d27-som1-ek-sd:~# reboot
```

See in the next **Listing 13-3** the "I2C to USB Multidisplay LED" driver source code (usb_ltc3206.c) for the SAMA5D27-SOM1 device.

Note: The source code of the driver and the Makefile can be downloaded from the GitHub repository of this book.

Listing 13-3: usb_ltc3206.c

```
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/usb.h>
#include <linux/i2c.h>
```

```
#define DRIVER_NAME       "usb-ltc3206"

#define USB_VENDOR_ID_LTC3206           0x04d8
#define USB_DEVICE_ID_LTC3206           0x003f

#define LTC3206_OUTBUF_LEN       3      /* USB write packet length */
#define LTC3206_I2C_DATA_LEN            3

/* Structure to hold all of our device specific stuff */
struct i2c_ltc3206 {
   u8 obuffer[LTC3206_OUTBUF_LEN];     /* USB write buffer */
   /* I2C/SMBus data buffer */
   u8 user_data_buffer[LTC3206_I2C_DATA_LEN];
   int ep_out;                  /* out endpoint */
   struct usb_device *usb_dev;  /* the usb device for this device */
   struct usb_interface *interface;/* the interface for this device */
   struct i2c_adapter adapter;  /* i2c related things */
   /* wq to wait for an ongoing write */
   wait_queue_head_t usb_urb_completion_wait;
   bool ongoing_usb_ll_op;             /* all is in progress */
   struct urb *interrupt_out_urb;
};

/*
 * Return list of supported functionality.
 */
static u32 ltc3206_usb_func(struct i2c_adapter *a)
{
   return I2C_FUNC_I2C | I2C_FUNC_SMBUS_EMUL |
          I2C_FUNC_SMBUS_READ_BLOCK_DATA | I2C_FUNC_SMBUS_BLOCK_PROC_CALL;
}

/* usb out urb callback function */
static void ltc3206_usb_cmpl_cbk(struct urb *urb)
{
   struct i2c_ltc3206 *dev = urb->context;
   int status = urb->status;
   int retval;

   switch (status) {
   case 0:               /* success */
          break;
   case -ECONNRESET:     /* unlink */
   case -ENOENT:
   case -ESHUTDOWN:
          return;
   /* -EPIPE:  should clear the halt */
```

```
    default:              /* error */
           goto resubmit;
    }

    /*
     * wake up the waiting function
     * modify the flag indicating the ll status
     */
    dev->ongoing_usb_ll_op = 0; /* communication is OK */
    wake_up_interruptible(&dev->usb_urb_completion_wait);
    return;

resubmit:
    retval = usb_submit_urb(urb, GFP_ATOMIC);
    if (retval) {
           dev_err(&dev->interface->dev,
                  "ltc3206(irq): can't resubmit intrerrupt urb, retval %d\n",
                  retval);
    }
}

static int ltc3206_ll_cmd(struct i2c_ltc3206 *dev)
{
    int rv;

    /*
     * tell everybody to leave the URB alone
     * we are going to write to the LTC3206 device
     */
    dev->ongoing_usb_ll_op = 1; /* doing USB communication */

    /* submit the interrupt out URB packet */
    if (usb_submit_urb(dev->interrupt_out_urb, GFP_KERNEL)) {
           dev_err(&dev->interface->dev,
                         "ltc3206(ll): usb_submit_urb intr out failed\n");
           dev->ongoing_usb_ll_op = 0;
           return -EIO;
    }

    /* wait for the transmit completion, the USB URB callback will signal it */
    rv = wait_event_interruptible(dev->usb_urb_completion_wait,
                  (!dev->ongoing_usb_ll_op));
    if (rv < 0) {
           dev_err(&dev->interface->dev, "ltc3206(ll): wait interrupted\n");
           goto ll_exit_clear_flag;
    }

    return 0;
```

```
ll_exit_clear_flag:
   dev->ongoing_usb_ll_op = 0;
   return rv;
}

static int ltc3206_init(struct i2c_ltc3206 *dev)
{
   int ret;

   /* initialize the LTC3206 */
   dev_info(&dev->interface->dev,
            "LTC3206 at USB bus %03d address %03d -- ltc3206_init()\n",
            dev->usb_dev->bus->busnum, dev->usb_dev->devnum);

   /* allocate the int out URB */
   dev->interrupt_out_urb = usb_alloc_urb(0, GFP_KERNEL);
   if (!dev->interrupt_out_urb){
          ret = -ENODEV;
          goto init_error;
   }

   /* Initialize the int out URB */
   usb_fill_int_urb(dev->interrupt_out_urb, dev->usb_dev,
                        usb_sndintpipe(dev->usb_dev,
                                       dev->ep_out),
                        (void *)&dev->obuffer, LTC3206_OUTBUF_LEN,
                        ltc3206_usb_cmpl_cbk, dev,
                        1);

   ret = 0;
   goto init_no_error;

init_error:
   dev_err(&dev->interface->dev, "ltc3206_init: Error = %d\n", ret);
   return ret;

init_no_error:
   dev_info(&dev->interface->dev, "ltc3206_init: Success\n");
   return ret;
}

static int ltc3206_i2c_write(struct i2c_ltc3206 *dev,
                                 struct i2c_msg *pmsg)
{
   u8 ucXferLen;
   int rv;
   u8 *pSrc, *pDst;
```

```
    if (pmsg->len > LTC3206_I2C_DATA_LEN)
    {
           pr_info ("problem with the lenght\n");
           return -EINVAL;
    }

    /* I2C write lenght */
    ucXferLen = (u8)pmsg->len;

    pSrc = &pmsg->buf[0];
    pDst = &dev->obuffer[0];
    memcpy(pDst, pSrc, ucXferLen);

    pr_info("oubuffer[0] = %d\n", dev->obuffer[0]);
    pr_info("oubuffer[1] = %d\n", dev->obuffer[1]);
    pr_info("oubuffer[2] = %d\n", dev->obuffer[2]);

    rv = ltc3206_ll_cmd(dev);
    if (rv < 0)
           return -EFAULT;

    return 0;
}

/* device layer, called from the I2C user app */
static int ltc3206_usb_i2c_xfer(struct i2c_adapter *adap,
           struct i2c_msg *msgs, int num)
{
    struct i2c_ltc3206 *dev = i2c_get_adapdata(adap);
    struct i2c_msg *pmsg;
    int ret, count;

    pr_info("number of i2c msgs is = %d\n", num);

    for (count = 0; count < num; count++) {
           pmsg = &msgs[count];
           ret = ltc3206_i2c_write(dev, pmsg);
           if (ret < 0)
                  goto abort;
    }

    /* if all the messages were transferred ok, return "num" */
    ret = num;
abort:
    return ret;
}
```

```
static const struct i2c_algorithm ltc3206_usb_algorithm = {
   .master_xfer = ltc3206_usb_i2c_xfer,
   .functionality = ltc3206_usb_func,
};

static const struct usb_device_id ltc3206_table[] = {
   { USB_DEVICE(USB_VENDOR_ID_LTC3206, USB_DEVICE_ID_LTC3206) },
   { }
};
MODULE_DEVICE_TABLE(usb, ltc3206_table);

static void ltc3206_free(struct i2c_ltc3206 *dev)
{
   usb_put_dev(dev->usb_dev);
   usb_set_intfdata(dev->interface, NULL);
   kfree(dev);
}

static int ltc3206_probe(struct usb_interface *interface,
                        const struct usb_device_id *id)
{
   struct usb_host_interface *hostif = interface->cur_altsetting;
   struct i2c_ltc3206 *dev;
   int ret;

   dev_info(&interface->dev, "ltc3206_probe() function is called.\n");

   /* allocate memory for our device and initialize it */
   dev = kzalloc(sizeof(*dev), GFP_KERNEL);
   if (dev == NULL) {
          pr_info("i2c-ltc3206(probe): no memory for device state\n");
          ret = -ENOMEM;
          goto error;
   }

   /* get ep_out address */
   dev->ep_out = hostif->endpoint[1].desc.bEndpointAddress;

   dev->usb_dev = usb_get_dev(interface_to_usbdev(interface));
   dev->interface = interface;

   init_waitqueue_head(&dev->usb_urb_completion_wait);

   /* save our data pointer in this interface device */
   usb_set_intfdata(interface, dev);

   /* setup I2C adapter description */
   dev->adapter.owner = THIS_MODULE;
```

```
    dev->adapter.class = I2C_CLASS_HWMON;
    dev->adapter.algo = &ltc3206_usb_algorithm;
    i2c_set_adapdata(&dev->adapter, dev);

    snprintf(dev->adapter.name, sizeof(dev->adapter.name),
             DRIVER_NAME " at bus %03d device %03d",
             dev->usb_dev->bus->busnum, dev->usb_dev->devnum);

    dev->adapter.dev.parent = &dev->interface->dev;

    /* initialize the ltc3206 device */
    ret = ltc3206_init(dev);
    if (ret < 0) {
            dev_err(&interface->dev, "failed to initialize adapter\n");
            goto error_init;
    }

    /* and finally attach to I2C layer */
    ret = i2c_add_adapter(&dev->adapter);
    if (ret < 0) {
            dev_info(&interface->dev, "failed to add I2C adapter\n");
            goto error_i2c;
    }

    dev_info(&dev->interface->dev,
                   "ltc3206_probe() -> chip connected -> Success\n");
    return 0;

error_init:
    usb_free_urb(dev->interrupt_out_urb);

error_i2c:
    usb_set_intfdata(interface, NULL);
    ltc3206_free(dev);
error:
    return ret;
}

static void ltc3206_disconnect(struct usb_interface *interface)
{
    struct i2c_ltc3206 *dev = usb_get_intfdata(interface);

    i2c_del_adapter(&dev->adapter);

    usb_kill_urb(dev->interrupt_out_urb);
    usb_free_urb(dev->interrupt_out_urb);
```

```
    usb_set_intfdata(interface, NULL);
    ltc3206_free(dev);

    pr_info("i2c-ltc3206(disconnect) -> chip disconnected");
}

static struct usb_driver ltc3206_driver = {
    .name = DRIVER_NAME,
    .probe = ltc3206_probe,
    .disconnect = ltc3206_disconnect,
    .id_table = ltc3206_table,
};

module_usb_driver(ltc3206_driver);

MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a usb controlled i2c ltc3206 device");
MODULE_LICENSE("GPL");
```

usb_ltc3206.ko Demonstration

```
/*
 * Connect the PIC32MX470 Curiosity Development Board USB Micro-B port (J12) to
 * the SAMA5D27-SOM1-EK1 J19 USB-B type C connector through a USB 2.0 Type-C male
 * to micro-B male cable. Power the SAMA5D27 board to boot the processor. Keep the
 * PIC32MX470 board powered off
 */

/* check the i2c adapters of the SAMA5D2 */
root@sama5d27-som1-ek-sd:~# i2cdetect -l
i2c-3   i2c             AT91                                    I2C adapter
i2c-1   i2c             AT91                                    I2C adapter
i2c-2   i2c             AT91                                    I2C adapter

root@sama5d27-som1-ek-sd:~# insmod usb_ltc3206.ko /* load the module */
usbcore: registered new interface driver usb-ltc3206

/* power now the PIC32MX Curiosity board */
root@sama5d27-som1-ek-sd:~# usb 2-2: new full-speed USB device number 2 using at
91_ohci
usb 2-2: New USB device found, idVendor=04d8, idProduct=003f
usb 2-2: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 2-2: Product: USB to I2C demo
usb 2-2: Manufacturer: Microchip Technology Inc.
usb-ltc3206 2-2:1.0: ltc3206_probe() function is called.
usb-ltc3206 2-2:1.0: LTC3206 at USB bus 002 address 002 -- ltc3206_init()
usb-ltc3206 2-2:1.0: ltc3206_init: Success
```

```
usb-ltc3206 2-2:1.0: ltc3206_probe() -> chip connected -> Success


/* check again the i2c adapters of the SAMA5D2, find the new one */
root@sama5d27-som1-ek-sd:~# i2cdetect -l
i2c-3   i2c             AT91                                    I2C adapter
i2c-1   i2c             AT91                                    I2C adapter
i2c-4   i2c             usb-ltc3206 at bus 002 device 002       I2C adapter
i2c-2   i2c             AT91                                    I2C adapter

root@sama5d27-som1-ek-sd:/sys/bus/usb/devices/2-2:1.0# ls
authorized         bInterfaceProtocol  ep_01     power
bAlternateSetting  bInterfaceSubClass  ep_81     subsystem
bInterfaceClass    bNumEndpoints       i2c-4     supports_autosuspend
bInterfaceNumber   driver              modalias  uevent

/*
 * verify the communication between the host and device
 * these commands toggle the three leds of the PIC32MX board and
 * set maximum brightness of the LTC3206 LED BLUE
 */
root@sama5d27-som1-ek-sd:~# i2cset -y 4 0x1b 0x00 0xf0 0x00 i
number of i2c msgs is = 1
oubuffer[0] = 0
oubuffer[1] = 240
oubuffer[2] = 0

/* set maximum brightness of the LTC3206 LED RED */
root@sama5d27-som1-ek-sd:~# i2cset -y 4 0x1b 0xf0 0x00 0x00 i

/* decrease brightness of the LTC3206 LED RED */
root@sama5d27-som1-ek-sd:~# i2cset -y 4 0x1b 0x10 0x00 0x00 i

/* set maximum brightness of the LTC3206 LED GREEN */
root@sama5d27-som1-ek-sd:~# i2cset -y 4 0x1b 0x00 0x0f 0x00 i

/* set maximum brightness of the LTC3206 LED GREEN and SUB display */
root@sama5d27-som1-ek-sd:~# i2cset -y 4 0x1b 0x00 0x0f 0x0f i

/* set maximum brightness of the LTC3206 MAIN display */
root@sama5d27-som1-ek-sd:~# i2cset -y 4 0x1b 0x00 0x00 0xf0 i

root@sama5d27-som1-ek-sd:~# rmmod usb_ltc3206.ko /* remove the module */
usbcore: deregistering interface driver usb-ltc3206

/* Power off the PIC32MX Curiosity board */
root@sama5d27-som1-ek-sd:~# i2c-ltc3206(disconnect) -> chip disconnected
usb 2-2: USB disconnect, device number 2
```

A

Porting Kernel Modules to the Microchip SAMA5D27-SOM1

Building an Embedded Linux System for the Microchip SAMA5D27-SOM1 System-On-Module

The SAMA5D2-SOM1 is a small single-sided System-On-Module (SOM) based on the high-performance 32-bit Arm® Cortex®-A5 processor-based MPU SAMA5D27 running up to 500 MHz. The SAMA5D27 SOM1 is built on a common set of proven Microchip components to reduce time to market by simplifying hardware design and software development. The SOM also simplifies design rules of the main application board, reducing overall PCB complexity and cost. You can check all the info related to this device at
https://www.microchip.com/wwwproducts/en/ATSAMA5D27-SOM1

The **SAMA5D27-SOM1-EK1** evaluation kit will be used for the development of the labs. The user guide and design files for this board can be found at
https://www.microchip.com/developmenttools/ProductDetails/atsama5d27-som1-ek1

Introduction

To get the Yocto Project expected behavior in a Linux Host Machine, the packages and utilities described below must be installed. An important consideration is the hard disk space required in the host machine. For example, when building on a machine running Ubuntu, the minimum hard disk space required is about 50 GB for the X11 backend. It is recommended that at least 120 GB is provided, which is enough to compile all backends together.

Host Packages

A Yocto Project build requires that some packages be installed for the build that are documented under the Yocto Project. Please make sure your host PC is running Ubuntu 16.04 64-bit and install the following packages:

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib \
```

```
build-essential chrpath socat cpio python python3 python3-pip python3-pexpect \
xz-utils debianutils iputils-ping libsdl1.2-dev xterm

$ sudo apt-get install autoconf libtool libglib2.0-dev libarchive-dev python-git \
sed cvs subversion coreutils texi2html docbook-utils python-pysqlite2 \
help2man make gcc g++ desktop-file-utils libgl1-mesa-dev libglu1-mesa-dev \
mercurial automake groff curl lzop asciidoc u-boot-tools dos2unix mtd-utils pv \
libncurses5 libncurses5-dev libncursesw5-dev libelf-dev zlib1g-dev
```

Yocto Project Setup and Image Building

The Yocto Project is a powerful building environment. It is build on top of several components including the famous OpenEmbedded build framework for embedded Linux. Poky is the reference system for building a whole embedded Linux distribution. The support for SAMA5 ARM® Cortex-A5-based MPUs has been added permitting to offer a root file system with a wide range of applications.

The support for the SAMA5 family is included in a particular Yocto layer: meta-atmel. The source for this layer are hosted on Linux4SAM GitHub account at https://github.com/linux4sam/meta-atmel

You can see the step-by-step build procedure below:

Create a directory:

```
~$ mkdir sama5d2_sumo
~$ cd sama5d2_sumo/
```

Clone poky git repository with the proper branch ready:

```
~/sama5d2_sumo$ git clone git://git.yoctoproject.org/poky -b sumo
```

Clone meta-openembedded git repository with the proper branch ready:

```
~/sama5d2_sumo$ git clone git://git.openembedded.org/meta-openembedded -b sumo
```

Clone meta-qt5 git repository with the proper branch ready:

```
~/sama5d2_sumo$ git clone git://code.qt.io/yocto/meta-qt5.git
~/sama5d2_sumo$ cd meta-qt5/
~/sama5d2_sumo/meta-qt5$ git checkout v5.9.6
~/sama5d2_sumo$ cd ..
~/sama5d2_sumo$
```

Clone meta-atmel layer with the proper branch ready:

```
~/sama5d2_sumo$ git clone git://github.com/linux4sam/meta-atmel.git -b sumo
```

Enter the poky directory to configure the build system and start the build process:

```
~/sama5d2_sumo$ cd poky/
~/sama5d2_sumo/poky$
```

Initialize the build directory:

```
~/sama5d2_sumo/poky$ source oe-init-build-env
```

Add meta-atmel layer to bblayer configuration file:

```
~/sama5d2_sumo/poky/build$ gedit conf/bblayers.conf

# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BSPDIR := "${@os.path.abspath(os.path.dirname(d.getVar('FILE', True)) +
'/../../..')}"

BBLAYERS ?= " \
  ${BSPDIR}/poky/meta \
  ${BSPDIR}/poky/meta-poky \
  ${BSPDIR}/poky/meta-yocto-bsp \
  ${BSPDIR}/meta-atmel \
  ${BSPDIR}/meta-openembedded/meta-oe \
  ${BSPDIR}/meta-openembedded/meta-networking \
  ${BSPDIR}/meta-openembedded/meta-python \
  ${BSPDIR}/meta-openembedded/meta-ruby \
  ${BSPDIR}/meta-openembedded/meta-multimedia \
  ${BSPDIR}/meta-qt5 \
  "

BBLAYERS_NON_REMOVABLE ?= " \
  ${BSPDIR}/poky/meta \
  ${BSPDIR}/poky/meta-poky \
  "
```

Edit local.conf to specify the machine, location of source archived, package type (rpm, deb or ipk). Set MACHINE name to "sama5d27-som1-ek-sd":

```
~/sama5d2_sumo/poky/build$ gedit conf/local.conf

[...]
MACHINE ??= "sama5d27-som1-ek-sd"
[...]
DL_DIR ?= "your_download_directory_path"
[...]
PACKAGE_CLASSES ?= "package_ipk"
[...]
USER_CLASSES ?= "buildstats image-mklibs"
```

To get better performance, use the "poky-atmel" distribution by also adding that line:

```
DISTRO = "poky-atmel"
```

Build the Atmel demo image. We found that additional local.conf changes are needed for our QT demo image. You can add these two lines at the end of the file:

```
~/sama5d2_sumo/poky/build$ gedit conf/local.conf

[...]
LICENSE_FLAGS_WHITELIST += "commercial"
SYSVINIT_ENABLED_GETTYS = ""

~/sama5d2_sumo/poky/build$ bitbake atmel-qt5-demo-image
```

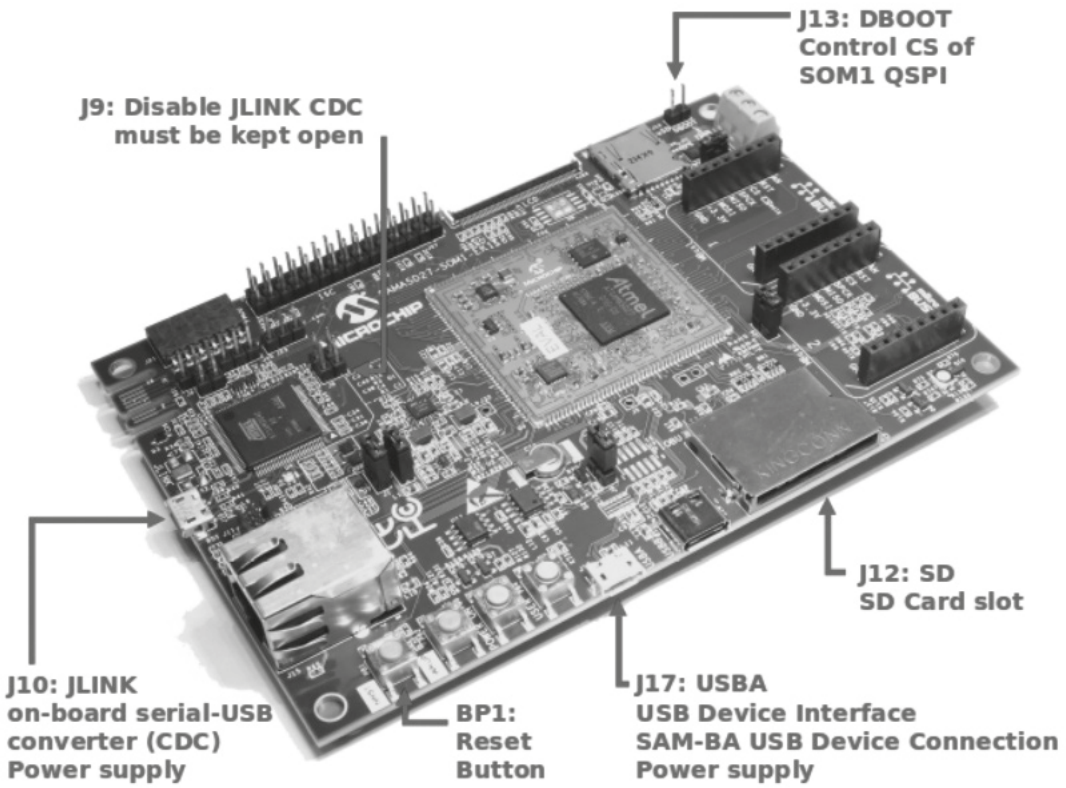
Enhancements are added on top of the official v4.14 Linux kernel tag where most of the Microchip SoC features are already supported. Note as well that Microchip re-integrate each and every stable kernel release on top of this Long Term Support (LTS) kernel revision. This means that each v4.14.x version is merged in Microchip branch. You will use the **linux4sam_6.0 tag** with integration of stable kernel updates up to **v4.14.73.** You can chek further information at
https://www.at91.com/linux4sam/bin/view/Linux4SAM/LinuxKernel

You are going to create a SD demo image compiled from **tag linux4sam_6.0**. Go to https://www.at91.com/linux4sam/bin/view/Linux4SAM/DemoArchive6_0 and download linux4sam-poky-sama5d27_som1_ek_video-6.0.img.bz2 yocto demo image.

To write the compressed image on the SD card, you will have to download and install **Etcher**. This tool, which is an Open Source software, is useful since it allows to get a compressed image as input. More information and extra help is available on the Etcher website at https://etcher.io/ Follow the steps of the Create a SD card with the demo section at https://www.at91.com/linux4sam/bin/view/Linux4SAM/Sama5d27Som1EKMainPage

Connect and Set Up Hardware

Connect the SAMA5D27-SOM1-EK1 board to your host computer by using a micro USB cable connected to the J10 JLink connector and open a serial console. Through this console, you can access to the SAMA5D27-SOM1-EK1 running an embedded Linux distribution.

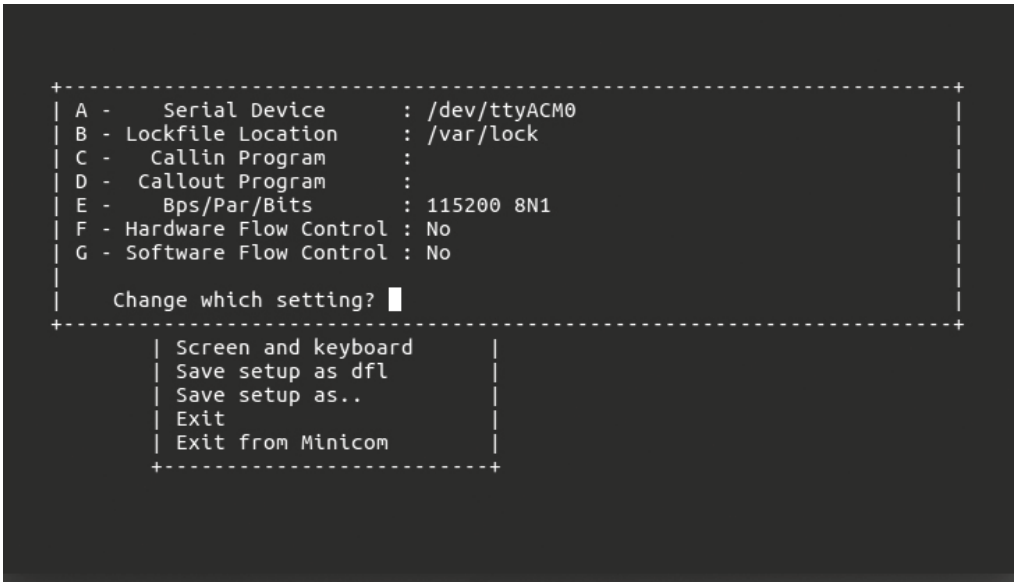


Launch a terminal on the host Linux PC by clicking on the Terminal icon. Type dmesg at the command prompt.

```
~$ dmesg
usb 3-1: New USB device found, idVendor=1366, idProduct=0106
usb 3-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
usb 3-1: Product: J-Link
usb 3-1: Manufacturer: SEGGER
usb 3-1: SerialNumber: 000483029459
cdc_acm 3-1:1.0: ttyACM0: USB ACM device
usbcore: registered new interface driver cdc_acm
cdc_acm: USB Abstract Control Model driver for USB modems and ISDN
adapters
```

In the log message you can see that the new USB device is found and installed as **ttyACM0**.

Launch and configure **minicom** in your host to see the booting of the system. Set the following configuration: "115.2 kbaud, 8 data bits, 1 stop bit, no parity".



Insert the SD card into the J12 SD slot on the SAMA5D27-SOM1-EK1 and reset the board (PB1/NRST). You should see Linux boot messages on the console.

Establish a network connection between the Linux host and the SAMA5D27-SOM1-EK target so that drivers built on the Linux Host PC can be easily transferred to the target for installation. Connect the Ethernet cable between your host PC and your SAMA5D27-SOM1-EK board. Set up the Linux host PC's IP Address to **10.0.0.1.** On the SAMA5D27-SOM1-EK board (the target), configure the eth0 interface with IP address **10.0.0.10** by editing the /etc/network/interfaces file. Open the file using vi editor:

```
root@sama5d27-som1-ek-sd:~# vi /etc/network/interfaces
```

Start editing the file by hitting "**i**". You may find there is already an entry for eth0. If not, configure eth0 with the following lines:

```
auto eth0
iface eth0 inet static
address 10.0.0.20
netmask 255.255.255.0
```

Exit the editing mode by hitting the ESC button. Save and exit the file by typing "**:wq**". If any time you want to exit the file without saving the changes you made, hit ESC followed by: "**:q!**".

Then, run the following commands to make sure that the eth0 Ethernet interface is properly configured:

```
root@sama5d27-som1-ek-sd:~# ifdown eth0
root@sama5d27-som1-ek-sd:~# ifup eth0
root@sama5d27-som1-ek-sd:~# ifconfig
```

Now, test that you can communicate with your Linux host machine from your SAMA5D27_SOM1_EK board. Exit the ping command by hitting **ctrl-C**.

```
root@sama5d27-som1-ek-sd:~# ping 10.0.0.1
```

A network connection is now established between the Linux host and the SAMA5D27_SOM1_EK target.

Working Outside of Yocto

In this section, we will describe the instructions to build the Yocto SDK for the ATSAMA5D27-SOM1 device:

```
~/sama5d2_sumo/poky/build$ bitbake -c populate_sdk atmel-qt5-demo-image
~/sama5d2_sumo/poky/build$ cd tmp/deploy/sdk/
~/sama5d2_sumo/poky/build/tmp/deploy/sdk$ ls
~/sama5d2_sumo/poky/build/tmp/deploy/sdk$ ./poky-atmel-glibc-x86_64-atmel-qt5-demo-
image-cortexa5hf-neon-toolchain-2.5.1.sh

Enter target directory for SDK (default: /opt/poky-atmel/2.5.1):
You are about to install the SDK to "/opt/poky-atmel/2.5.1". Proceed[Y/n]?
Extracting SDK.....................................................................
...............done
Setting it up...done
SDK has been successfully set up and is ready to be used.
```

Each time you wish to use the SDK in a new shell session, you need to source the environment setup script:

```
~$ . /opt/poky-atmel/2.5.1/environment-setup-cortexa5hf-neon-poky-linux-gnueabi
```

Building the Linux Kernel

In this section, we will describe the instructions to build the Linux kernel for the ATSAMA5D27-SOM1 device:

Download the kernel sources from the Microchip git:

```
~$ git clone git://github.com/linux4sam/linux-at91.git
~$ cd linux-at91/
~/linux-at91$ git branch -r
~/linux-at91$ git checkout origin/linux-4.14-at91 -b linux-4.14-at91
```

Compile the kernel image, modules, and all the device tree files:

```
~/linux-at91$ make mrproper
~/linux-at91$ make ARCH=arm sama5_defconfig
~/linux-at91$ make ARCH=arm menuconfig
```

Configure the following kernel settings that will be needed during the development of the drivers:

```
Device drivers >
   [*] SPI support  --->
          <*>   User mode SPI device driver support

Device drivers >
   [*] LED Support  --->
           <*>   LED Class Support
           -*-   LED Trigger support  --->
                        <*>   LED Timer Trigger
                        <*>   LED Heartbeat Trigger

Device drivers >
   <*> Industrial I/O support  --->
          -*-   Enable buffer support within IIO
          -*-   Industrial I/O buffering based on kfifo
          <*>  Enable IIO configuration via configfs
          -*-   Enable triggered sampling support
          <*>   Enable software IIO device support
          <*>   Enable software triggers support
                 Triggers - standalone  --->
                          <*> High resolution timer trigger
                          <*> SYSFS trigger

Device drivers >
    <*> Userspace I/O drivers  --->
          <*>   Userspace I/O platform driver with generic IRQ handling
          <*>   Userspace platform driver with generic irq and dynamic memory

Device drivers >
   Input device support  --->
          -*- Generic input layer (needed for keyboard, mouse, ...)
          <*>   Polled input device skeleton
          <*>   Event interface
```

Save the configuration and exit from menuconfig.

Source the toolchain script and compile the kernel, device tree files and modules in a single step:

```
~/linux-at91$ source /opt/poky-atmel/2.5.1/environment-setup-cortexa5hf-neon-
poky-linux-gnueabi
~/linux-at91$ make -j4
```

Once the Linux kernel, dtb files and modules have been compiled they must be installed. In the case of the kernel image this can be installed by copying the zImage file to the location where it will be read from. There are two partitions on the SD card. The smaller one, formatted as a FAT file system, holds boot.bin, u-boot.bin, uboot.env, and the FIT image **sama5d27_som1_ek.itb**. The larger partition, formatted as ETX4, contains a root file system. The SD card as whole is represented among the devices as /dev/mmcblk0, its partitions as /dev/mmcblk0p1 and /dev/mmcblk0p2 respectively. The second partition is mounted as a root on the target. The boot partition is not mounted. We must mount it first before we can access it.

Starting from **U-boot 2018.07** released in **Linux4SAM6.0**, we can use the feature of patching the Device Tree Blob (DTB) with additional Device Tree Overlays (DTBO). A device tree overlay is a file that can be used at runtime (by the bootloader in our case) to dynamically modify the device tree, adding nodes to the tree and making changes to properties in the existing tree.

You can easily download the DT Overlay source code from the Linux4SAM GitHub DT Overlays repository. Clone the Linux4sam GitHub DT Overlay repository:

```
~$ git clone git://github.com/linux4sam/dt-overlay-at91.git
   Cloning into 'dt-overlay-at91'...
   remote: Enumerating objects: 760, done.
   remote: Counting objects: 100% (760/760), done.
   remote: Compressing objects: 100% (428/428), done.
   remote: Total 760 (delta 340), reused 735 (delta 315), pack-reused 0
   Receiving objects: 100% (760/760), 369.55 KiB | 1.23 MiB/s, done.
   Resolving deltas: 100% (340/340), done.

~$ cd dt-overlay-at91/
```

The source code has been taken from the master branch which is pointing to the latest branch we use.

Now, build the DT-Overlay and the FIT image. The FIT image is a placeholder that has the zImage and the base Device Tree, plus additional overlays that can be selected at boot time. To build the overlays and the FIT image with overlays for a board make sure the following steps are done:

- the environment variables ARCH and CROSS_COMPILE are set correctly
- (optional) the environment variable KERNEL_DIR points to Linux kernel and the kernel was already built for the board. This is needed because the DT Overlay repository uses the Device Tree Compiler (dtc) from the kernel source tree. By default, KERNEL_DIR is set to a linux directory that would be under the parent directory in the directory tree: ../linux
- (optional) the environment variable KERNEL_BUILD_DIR that points to where the Linux kernel binary and Device Tree blob, resulting of your compilation of the kernel, are located. By default, KERNEL_BUILD_DIR is set to the same directory as KERNEL_DIR. It shouldn't be changed if you have the habit of compiling your kernel within the Linux source tree
- mkimage is installed on the development machine
- the Device Tree Compiler from Linux kernel is in the PATH environment variable

```
~/dt-overlay-at91$ source /opt/poky-atmel/2.5.1/environment-setup-cortexa5hf-neon-
poky-linux-gnueabi

~/dt-overlay-at91$ gedit Makefile
#KERNEL_DIR?=../linux
KERNEL_DIR?=/home/alberto/linux-at91

~/dt-overlay-at91$ make sama5d27_som1_ek_dtbos
~/dt-overlay-at91$ make sama5d27_som1_ek.itb
```

Now, on the target mount /dev/mmcblk0p1 on /mnt. The /mnt is an empty directory that is frequently used to mount external file systems.

```
root@sama5d27-som1-ek-sd:~# ls -lF /mnt
total 0
root@sama5d27-som1-ek-sd:~# mount /dev/mmcblk0p1 /mnt
root@sama5d27-som1-ek-sd:~# ls -lF /mnt/
total 4278
-rwxr-xr-x 1 root root   19141 Oct 12  2018 BOOT.BIN
-rwxr-xr-x 1 root root 3901236 Oct 12  2018 sama5d27_som1_ek.itb
-rwxr-xr-x 1 root root  442415 Oct 12  2018 u-boot.bin
-rwxr-xr-x 1 root root   16384 Oct 12  2018 uboot.env
```

Copy the FIT from the host to target's /mnt directory. We use **scp** (secure copy) program. We copy the file as root, because we logged in to the target as root. Enter the commands on the host terminal. When a secure copy program runs for the first time, it will notice it is communicating with an unknown machine. It will report the fact. Accept it. The host PC will add the target to the list of known machines.

```
~/dt-overlay-at91$ scp sama5d27_som1_ek.itb root@10.0.0.10:/mnt
```

```
root@sama5d27-som1-ek-sd:~# umount /mnt
root@sama5d27-som1-ek-sd:~# reboot
```

For the SAMA5Dx based boards, Microchip implements a common dtsi files, which define peripherals present on a board, a SoM and a SoC. Each variant of SAMA5Dx-EK or Xplained board has its own dts file. For example, the at91-sama5d27_som1_ek.dts, which defines a plain board, includes the at91-sama5d27_som1_ek_common.dtsi that enables the board's common peripherals. It includes the at91-sama5d27_som1.dtsi file that defines the peripherals present on the SoM. This file in turn includes the sama5d2.dtsi file, which defines all peripherals present on the SoC's, but sets their status to "disabled" except for the three crypto devices. The top dts files provide the final touch enabling extra devices and creating through it different flavors of the board. The device tree source files (.dts and .dtsi) for ARM core based devices in Linux kernel are stored together with Linux kernel sources in linux-at91/arch/arm/boot/dts directory. You can also find them on Linux4SAM github at
https://github.com/linux4sam/linux-at91/tree/master/arch/arm/boot/dts

The at91-sama5d27_som1_ek.dts is the Device Tree file for the SAMA5D27-SOM1-EK board. Every time you add a new node or modify something in this DT source file you have to execute the following steps to make the new dtb file available on your target device:

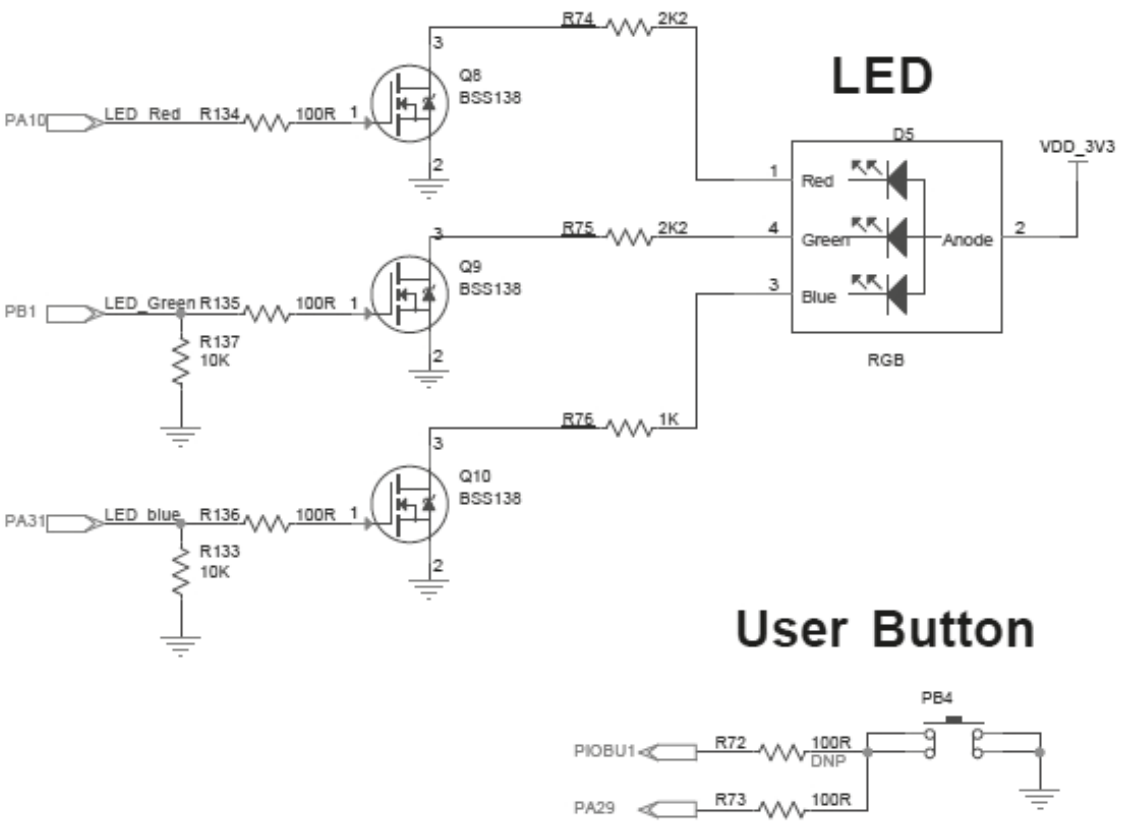
```
~/linux-at91$ make dtbs
~/dt-overlay-at91$ make sama5d27_som1_ek.itb
root@sama5d27-som1-ek-sd:~# mount /dev/mmcblk0p1 /mnt
~/dt-overlay-at91$ scp sama5d27_som1_ek.itb root@10.0.0.10:/mnt
root@sama5d27-som1-ek-sd:~# umount /mnt
root@sama5d27-som1-ek-sd:~# reboot
```

Hardware Descriptions for the Microchip SAMA5D27-SOM1 System-On-Module Labs

In the following sections, we will describe the SAMA5D27-SOM1-EK hardware settings for the development of the labs.

LAB 5.2, 5.3 and 5.4 Hardware Descriptions

The SAMA5D27-SOM1-EK board integrates a RGB LED. Go to the pag.8 of the schematics to see it. The tricolor LED (D5) contains three RGB LEDs in the same housing. Each LED is individually controlled by a processor pin programmed as a GPIO output. These pins are PA10, PA31, PB1. Currently, these pins are used by the "leds" driver, therefore you'll have to disable it inside the device tree.

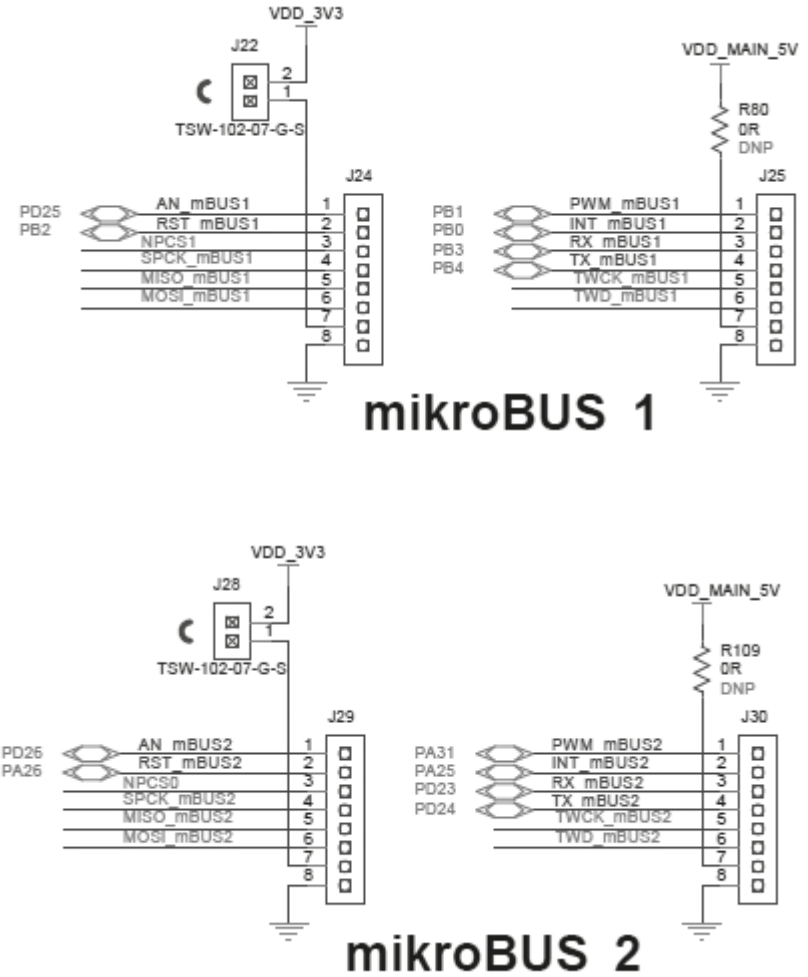


LAB 6.1 Hardware Description

In this lab, the driver will control several PCF8574 I/O expander devices connected to the I2C bus. You can use one the multiples boards based on this device to develop this lab, for example, the following one https://www.waveshare.com/pcf8574-io-expansion-board.htm

The SAMA5D27-SOM1-EK board features a wide range of peripherals, as well as an user interface and expansion options, including two mikroBUS™ click interface headers. The mikroBUS standard defines the main board sockets and add-on boards (a.k.a. "click boards") used for interfacing microprocessors with integrated modules with proprietary pin configuration and silkscreen markings. The pinout consists of three groups of communication pins (SPI, UART and TWI), four additional pins (PWM, interrupt, analog input and reset) and

two power groups (+3.3V and GND on the left, and 5V and GND on the right 1x8 header). Go to the pag.9 of the SAMA5D27-SOM1-EK board schematics to see both connectors.

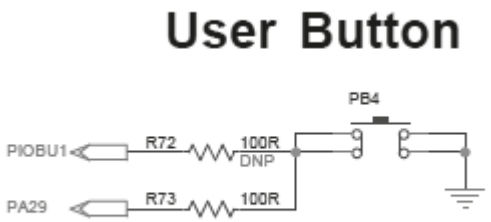


You can use two PCF8574 boards, one connected to the I2C TWCK_mBUS1 and TWD_mBUS1 signals and another connected to the I2C TWCK_mBUS2 and TWD_mBUS2 signals. Don´t forget to connect 3V3 and GND signals between each PCF8574 board and its respective mikroBUS connector.

LAB 6.2 Hardware Description

To test this driver, you will use the DC749A - Demo Board (http://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/dc749a.html).

In this lab, you will connect the I2C pins between the SAMA5D27-SOM1-EK mikroBUS 1 connector and the DC749A - Demo Board. Go to the pag.9 of the SAMA5D27-SOM1-EK board schematics to find the mikroBUS 1 connector and look for the TWCK_mBUS1 and TWD_mBUS1 pins. Connect the mikroBUS TWD pin to the pin 7 (SDA) of the DC749A J1 connector and the mikroBUS TWCK pin to the pin 4 (SCL) of the DC749A J1 connector. Connect the mikroBUS 3.3V pin to the DC749A Vin J2 pin and to the DC749A J20 DVCC pin. Connect the mikroBUS AN_mBUS1 pin (PD25 pad) to the pin 6 (ENRGB/S) of the DC749A J1 connector. Do not forget to connect GND between the two boards.



LAB 7.1 and 7.2 Hardware Descriptions

In these two labs, you will use the "USER" button (PB4). This button is connected to the PA29 pin. This pin is used by the "gpio_keys" driver, so it must be disabled inside the device tree. The pin will be programmed as an input and will generate an interrupt. You will also have to ensure the mechanical key is debounced. Open the SAMA5D27-SOM1-EK board schematics and find the button PB4 in the pag.8.

LAB 7.3 Hardware Description

The SAMA5D27-SOM1-EK board integrates a RGB LED. Go to the pag.8 of the schematics to see it. The tricolor LED (D5) contains three RGB LEDs in the same housing. Each LED is individually controlled by a processor pin programmed as a GPIO output. These pins are PA10, PA31, PB1. Currently, these pins are used by the "leds" driver, therefore you'll have to disable it inside the device tree. In this lab, you will also use the "USER" button (PB4). This button is connected to the PA29 pin. This pin is used

by the "gpio_keys" driver, so it must be disabled inside the device tree. The pin will be programmed as an input and will generate an interrupt. You will also have to ensure the mechanical key is debounced. Open the schematics and find the PB4 button in the pag.8.

A second interrupt will be generated by using the button of the **MikroElektronika Button R click board.** You can see the board at https://www.mikroe.com/button-r-click. You can download the schematics from that link or from the GitHub repository of this book. Connect this board to the SAMA5D27-SOM1-EK mikroBUS 1 connector. The pin INT_mBUS1 (pad PB0) of the mikroBUS 1 will be programmed as an input and will generate a second interrupt.

LAB 10.1,10.2 and 12.1 Hardware Descriptions

In these labs, you will control an accelerometer board connected to the I2C and SPI buses of the processor. You will use the ADXL345 Accel click mikroBUS™ accessory board to develop the drivers; you can download the schematics for the board at http://www.mikroe.com/click/accel/

Connect the board to the SAMA5D27-SOM1-EK mikroBUS 2 connector. In labs 10.2 and 12.1 the INT_mBUS2 pin (PA25 pad) of the mikroBUS 2 connector will be programmed as an input and will generate an interrupt.

LAB 11.1 Hardware Description

In this lab, you will control the Analog Devices LTC2607 internal DACs individually or both DACA + DACB in a simultaneous mode. You will use the DC934A evaluation board; you can download its schematics at https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/dc934a.html

You will connect the TWCK_mBUS2 and TWD_mBUS2 pins of the SAMA5D27-SOM1-EK mikroBUS 2 connector to the SDA and SCL pins of the LTC2607 DC934A evaluation board. You will power the LTC2607 device by connecting the mikroBUS 3.3V pin to the V+, pin 1 of the DC934A connector J1. You will also connect GND between the DC934A (i.e., pin 3 of connector J1) board and the mikroBUS 2 GND pin of the SAMA5D27-SOM1-EK board.

LAB 11.2 and LAB 11.3 Hardware Descriptions

In these two labs, you will reuse the hardware settings of the lab 11.1 and will connect the SPI pins of the SAMA5D27-SOM1-EK mikroBUS 2 to the SPI ones of the LTC2422 dual ADC SPI device that is included in the DC934A board.

Open the SAMA5D27-SOM1-EK board schematics and find the SPI pins of the mikroBUS 2 connector. You will use the CS, SCK and MISO (Master In, Slave Out) signals. The MOSI (Master out, Slave in) signal won´t be needed, as you are only going to receive data from the LTC2422

device. Connect the following mikroBUS 2 pins to the LTC2422 SPI ones, which are available from the J1 connector of the DC934A board:

- Connect SAMA5D27-SOM1 **NPCS0** (CS) pin to LTC2422 **CS** pin
- Connect SAMA5D27-SOM1 **SPCK_mBUS2** (SCK) pin to LTC2422 **SCK** pin
- Connect SAMA5D27-SOM1 **MISO_mBUS2** (MISO) pin to LTC2422 **MISO** pin

In the lab 11.3, you will also use the "USER" button (PB4). This button is connected to the PA29 pin. This pin is used by the "gpio_keys" driver, so it must be disabled inside the device tree. The pin will be programmed as an input and will generate an interrupt.

The SAMA5D27-SOM1 kernel modules, applications and device tree settings are included in the linux_4.14_sama5d27-SOM_drivers.zip file that can be downloaded from the GitHub repository at https://github.com/ALIBERA/linux_book_2nd_edition

References

1. NXP, "i.MX 7Dual Applications Processor Reference Manual". Document Number: IMX7DRM Rev. 0.1, 08/2016.

2. Microchip, "SAMA5D2 Series Datasheet". Datasheet number: DS60001476B.

3. Broadcom, "BCM2835 ARM Peripherals guide".

4. NXP, "i.MX Yocto Project User's Guide". Document Number: IMXLXYOCTOUG. Rev. L4.9.11_1.0.0-ga+mx8-alpha, 09/2017.

5. Marcin Bis, "Exploring Linux Kernel Source Code with Eclipse and QTCreator". ELCE 2016, Berlin, 10/2016.
https://bis-linux.com/en/elc_europe2016

6. Raspberry Pi Linux documentation.
https://www.raspberrypi.org/documentation/linux/

7. Linux & Open Source related information for AT91 Smart ARM Microcontrollers.
http://www.at91.com/linux4sam/bin/view/Linux4SAM

8. The Linux kernel DMAEngine documentation.
https://www.kernel.org/doc/html/v4.15/driver-api/dmaengine/index.html

9. The Linux kernel Input documentation.
https://www.kernel.org/doc/html/v4.12/input/input.html

10. The Linux kernel PINCTRL (PIN CONTROL) subsystem documentation.
https://www.kernel.org/doc/html/v4.15/driver-api/pinctl.html

11. The Linux kernel General Purpose Input/Output (GPIO) documentation.
https://www.kernel.org/doc/html/v4.17/driver-api/gpio/index.html

12. Hans-Jürgen Koch, "The Userspace I/O HOWTO".
http://www.hep.by/gnu/kernel/uio-howto/

13. Daniel Baluta, "Industrial I/O driver developer's guide".
https://dbaluta.github.io/

14. Rubini, Corbet, and Kroah-Hartman, "Linux Device Drivers", third edition, O'Reilly, 02/2005.
https://lwn.net/Kernel/LDD3/

15. bootlin, "Linux Kernel and Driver Development Training".
https://bootlin.com/doc/training/linux-kernel/

16. Corbet, LWN.net, Kroah-Hartman, The Linux Foundation, "Linux Kernel Development report", seventh edition, 08/2016.
https://www.linux.com/publications/linux-kernel-development-how-fast-it-going-who-doing-it-what-they-are-doing-and-who-5/

17. Silicon Labs, "Human Interface Device Tutorial". Application note number: AN249
https://www.silabs.com/documents/public/application-notes/AN249.pdf

18. Exar, "USB Basics for the EXAR Family of USB Uarts". Application note number: AN213
https://www.exar.com/appnote/an213.pdf

19. The Linux kernel USB API documentation.
https://www.kernel.org/doc/html/v4.14/driver-api/usb/index.html

Index

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y