Linux UWB + Wireless USB + WiNET

Inaky Perez-Gonzalez <inaky.perez-gonzalez@intel.com>

Please visit http://bughost.org/thewiki/Design-overview.txt-1.8 for
updated content.

* Design-overview.txt-1.8

This code implements a Ultra Wide Band stack for Linux, as well as
drivers for the the USB based UWB radio controllers defined in the
Wireless USB 1.0 specification (including Wireless USB host controller
and an Intel WiNET controller).

Introduction

UWB is a wide-band communication protocol that is to serve also as the

low-level protocol for others (much like TCP sits on IP). Currently these others are Wireless USB and TCP/IP, but seems Bluetooth and Firewire/1394 are coming along.

UWB uses a band from roughly 3 to 10 GHz, transmitting at a max of ~-41dB (or 0.074 uW/MHz--geography specific data is still being negotiated w/ regulators, so watch for changes). That band is divided in a bunch of ~1.5 GHz wide channels (or band groups) composed of three subbands/subchannels (528 MHz each). Each channel is independent of each other, so you could consider them different "busses". Initially this driver considers them all a single one.

Radio time is divided in 65536 us long /superframes/, each one divided in 256 256us long /MASs/ (Media Allocation Slots), which are the basic time/media allocation units for transferring data. At the beginning of each superframe there is a Beacon Period (BP), where every device transmit its beacon on a single MAS. The length of the BP depends on how many devices are present and the length of their beacons.

Devices have a MAC (fixed, 48 bit address) and a device (changeable, 16 bit address) and send periodic beacons to advertise themselves and pass info on what they are and do. They advertise their capabilities and a bunch of other stuff.

The different logical parts of this driver are:

*

  UWB: the Ultra-Wide-Band stack -- manages the radio and associated spectrum to allow for devices sharing it. Allows to control bandwidth assignment, beaconing, scanning, etc

*

  WUSB: the layer that sits on top of UWB to provide Wireless USB. The Wireless USB spec defines means to control a UWB radio and to do the actual WUSB.

  HWA: Host Wire adapters, your Wireless USB dongle

WUSB also defines a device called a Host Wire Adaptor (HWA), which in mere terms is a USB dongle that enables your PC to have UWB and Wireless USB. The Wireless USB Host Controller in a HWA looks to the host like a [Wireless] USB controller connected via USB (!)

The HWA itself is broken in two or three main interfaces:

*

  RC: Radio control -- this implements an interface to the Ultra-Wide-Band radio controller. The driver for this implements a USB-based UWB Radio Controller to the UWB stack.

*

HC: the wireless USB host controller. It looks like a USB host whose root port is the radio and the WUSB devices connect to it. To the system it looks like a separate USB host. The driver (will) implement a USB host controller (similar to UHCI, OHCI or EHCI) for which the root hub is the radio...To reiterate: it is a USB controller that is connected via USB instead of PCI.

*

WINET: some HW provide a WiNET interface (IP over UWB). This package provides a driver for it (it looks like a network interface, winetX). The driver detects when there is a link up for their type and kick into gear.

DWA: Device Wired Adaptor, a Wireless USB hub for wired devices

These are the complement to HWAs. They are a USB host for connecting wired devices, but it is connected to your PC connected via Wireless USB. To the system it looks like yet another USB host. To the untrained eye, it looks like a hub that connects upstream wirelessly.

We still offer no support for this; however, it should share a lot of code with the HWA-RC driver; there is a bunch of factorization work that has been done to support that in upcoming releases.

WHCI: Wireless Host Controller Interface, the PCI WUSB host adapter

This is your usual PCI device that implements WHCI. Similar in concept to EHCI, it allows your wireless USB devices (including DWAs) to connect to your host via a PCI interface. As in the case of the HWA, it has a Radio Control interface and the WUSB Host Controller interface per se.

There is still no driver support for this, but will be in upcoming releases.

The UWB stack

The main mission of the UWB stack is to keep a tally of which devices are in radio proximity to allow drivers to connect to them. As well, it provides an API for controlling the local radio controllers (RCs from now on), such as to start/stop beaconing, scan, allocate bandwidth, etc.

Devices and hosts: the basic structure

The main building block here is the UWB device (struct uwb_dev). For each device that pops up in radio presence (ie: the UWB host receives a beacon from it) you get a struct uwb_dev that will show up in /sys/class/uwb and in /sys/bus/uwb/devices.

For each RC that is detected, a new struct uwb_rc is created. In turn, a RC is also a device, so they also show in /sys/class/uwb and /sys/bus/uwb/devices, but at the same time, only radio controllers show

up in /sys/class/uwb_rc.

*

[*] The reason for RCs being also devices is that not only we can see them while enumerating the system device tree, but also on the radio (their beacons and stuff), so the handling has to be likewise to that of a device.

Each RC driver is implemented by a separate driver that plugs into the interface that the UWB stack provides through a struct uwb_rc_ops. The spec creators have been nice enough to make the message format the same for HWA and WHCI RCs, so the driver is really a very thin transport that moves the requests from the UWB API to the device [/uwb_rc_ops->cmd()/] and sends the replies and notifications back to the API [/uwb_rc_neh_grok()/]. Notifications are handled to the UWB daemon, that is chartered, among other things, to keep the tab of how the UWB radio neighborhood looks, creating and destroying devices as they show up or disappear.

Command execution is very simple: a command block is sent and a event block or reply is expected back. For sending/receiving command/events, a handle called /neh/ (Notification/Event Handle) is opened with /uwb_rc_neh_open()/.

The HWA-RC (USB dongle) driver (drivers/uwb/hwa-rc.c) does this job for the USB connected HWA. Eventually, drivers/whci-rc.c will do the same for the PCI connected WHCI controller.

Host Controller life cycle

So let's say we connect a dongle to the system: it is detected and firmware uploaded if needed [for Intel's i1480 /drivers/uwb/ptc/usb.c:ptc_usb_probe()/] and then it is reenumerated. Now we have a real HWA device connected and /drivers/uwb/hwa-rc.c:hwarc_probe()/ picks it up, that will set up the Wire-Adaptor environment and then suck it into the UWB stack's vision of the world [/drivers/uwb/lc-rc.c:uwb_rc_add()/].

*

[*] The stack should put a new RC to scan for devices [/uwb_rc_scan()/] so it finds what's available around and tries to connect to them, but this is policy stuff and should be driven from user space. As of now, the operator is expected to do it manually; see the release notes for documentation on the procedure.

When a dongle is disconnected, /drivers/uwb/hwa-rc.c:hwarc_disconnect()/ takes time of tearing everything down safely (or not...).

On the air: beacons and enumerating the radio neighborhood

So assuming we have devices and we have agreed for a channel to connect on (let's say 9), we put the new RC to beacon:

*

```
$ echo 9 0 > /sys/class/uwb_rc/uwb0/beacon
```

Now it is visible. If there were other devices in the same radio channel and beacon group (that's what the zero is for), the dongle's radio control interface will send beacon notifications on its notification/event endpoint (NEEP). The beacon notifications are part of the event stream that is funneled into the API with /drivers/uwb/neh.c:uwb_rc_neh_grok()/ and delivered to the UWBD, the UWB daemon through a notification list.

UWBD wakes up and scans the event list; finds a beacon and adds it to the BEACON CACHE (/uwb_beca/). If he receives a number of beacons from the same device, he considers it to be 'onair' and creates a new device [/drivers/uwb/lc-dev.c:uwbd_dev_onair()/]. Similarly, when no beacons are received in some time, the device is considered gone and wiped out [uwbd calls periodically /uwb/beacon.c:uwb_beca_purge()/ that will purge the beacon cache of dead devices].

Device lists

All UWB devices are kept in the list of the struct bus_type uwb_bus.

Bandwidth allocation

The UWB stack maintains a local copy of DRP availability through processing of incoming *DRP Availability Change* notifications. This local copy is currently used to present the current bandwidth availability to the user through the sysfs file /sys/class/uwb_rc/uwbx/bw_avail. In the future the bandwidth availability information will be used by the bandwidth reservation routines.

The bandwidth reservation routines are in progress and are thus not present in the current release. When completed they will enable a user to initiate DRP reservation requests through interaction with sysfs. DRP reservation requests from remote UWB devices will also be handled. The bandwidth management done by the UWB stack will include callbacks to the higher layers will enable the higher layers to use the reservations upon completion. [Note: The bandwidth reservation work is in progress and subject to change.]

Wireless USB Host Controller drivers

WARNING This section needs a lot of work!

As explained above, there are three different types of HCs in the WUSB world: HWA-HC, DWA-HC and WHCI-HC.

HWA-HC and DWA-HC share that they are Wire-Adapters (USB or WUSB connected controllers), and their transfer management system is almost

identical. So is their notification delivery system.

HWA-HC and WHCI-HC share that they are both WUSB host controllers, so they have to deal with WUSB device life cycle and maintenance, wireless root-hub

HWA exposes a Host Controller interface (HWA-HC 0xe0/02/02). This has three endpoints (Notifications, Data Transfer In and Data Transfer Out--known as NEP, DTI and DTO in the code).

We reserve UWB bandwidth for our Wireless USB Cluster, create a Cluster ID and tell the HC to use all that. Then we start it. This means the HC starts sending MMCs.

*

  The MMCs are blocks of data defined somewhere in the WUSB1.0 spec that define a stream in the UWB channel time allocated for sending WUSB IEs (host to device commands/notifications) and Device Notifications (device initiated to host). Each host defines a unique Wireless USB cluster through MMCs. Devices can connect to a single cluster at the time. The IEs are Information Elements, and among them are the bandwidth allocations that tell each device when can they transmit or receive.

Now it all depends on external stimuli.

New device connection

A new device pops up, it scans the radio looking for MMCs that give out the existence of Wireless USB channels. Once one (or more) are found, selects which one to connect to. Sends a /DN_Connect/ (device notification connect) during the DNTS (Device Notification Time Slot--announced in the MMCs

HC picks the /DN_Connect/ out (nep module sends to notif.c for delivery into /devconnect/). This process starts the authentication process for the device. First we allocate a /fake port/ and assign an unauthenticated address (128 to 255--what we really do is 0x80 | fake_port_idx). We fiddle with the fake port status and /khubd/ sees a new connection, so he moves on to enable the fake port with a reset.

So now we are in the reset path -- we know we have a non-yet enumerated device with an unauthorized address; we ask user space to authenticate (FIXME: not yet done, similar to bluetooth pairing), then we do the key exchange (FIXME: not yet done) and issue a /set address 0/ to bring the device to the default state. Device is authenticated.

From here, the USB stack takes control through the usb_hcd ops. khubd has seen the port status changes, as we have been toggling them. It will start enumerating and doing transfers through usb_hcd->urb_enqueue() to read descriptors and move our data.

Device life cycle and keep alives

Every time there is a successful transfer to/from a device, we update a

per-device activity timestamp. If not, every now and then we check and if the activity timestamp gets old, we ping the device by sending it a Keep Alive IE; it responds with a /DN_Alive/ pong during the DNTS (this arrives to us as a notification through devconnect.c:wusb_handle_dn_alive(). If a device times out, we disconnect it from the system (cleaning up internal information and toggling the bits in the fake hub port, which kicks khubd into removing the rest of the stuff).

This is done through devconnect:__wusb_check_devs(), which will scan the device list looking for whom needs refreshing.

If the device wants to disconnect, it will either die (ugly) or send a /DN_Disconnect/ that will prompt a disconnection from the system.

Sending and receiving data

Data is sent and received through /Remote Pipes/ (rpipes). An rpipe is /aimed/ at an endpoint in a WUSB device. This is the same for HWAs and DWAs.

Each HC has a number of rpipes and buffers that can be assigned to them; when doing a data transfer (xfer), first the rpipe has to be aimed and prepared (buffers assigned), then we can start queueing requests for data in or out.

Data buffers have to be segmented out before sending--so we send first a header (segment request) and then if there is any data, a data buffer immediately after to the DTI interface (yep, even the request). If our buffer is bigger than the max segment size, then we just do multiple requests.

[This sucks, because doing USB scatter gatter in Linux is resource intensive, if any...not that the current approach is not. It just has to be cleaned up a lot :)].

If reading, we don't send data buffers, just the segment headers saying we want to read segments.

When the xfer is executed, we receive a notification that says data is ready in the DTI endpoint (handled through xfer.c:wa_handle_notif_xfer()). In there we read from the DTI endpoint a descriptor that gives us the status of the transfer, its identification (given when we issued it) and the segment number. If it was a data read, we issue another URB to read into the destination buffer the chunk of data coming out of the remote endpoint. Done, wait for the next guy. The callbacks for the URBs issued from here are the ones that will declare the xfer complete at some point and call its callback.

Seems simple, but the implementation is not trivial.

    *

      WARNING Old!!

The main xfer descriptor, wa_xfer (equivalent to a URB) contains an

array of segments, tallys on segments and buffers and callback information. Buried in there is a lot of URBs for executing the segments and buffer transfers.

For OUT xfers, there is an array of segments, one URB for each, another one of buffer URB. When submitting, we submit URBs for segment request 1, buffer 1, segment 2, buffer 2...etc. Then we wait on the DTI for xfer result data; when all the segments are complete, we call the callback to finalize the transfer.

For IN xfers, we only issue URBs for the segments we want to read and then wait for the xfer result data.

URB mapping into xfers

This is done by hwahc_op_urb_[en|de]queue(). In enqueue() we aim an rpipe to the endpoint where we have to transmit, create a transfer context (wa_xfer) and submit it. When the xfer is done, our callback is called and we assign the status bits and release the xfer resources.

In dequeue() we are basically cancelling/aborting the transfer. We issue a xfer abort request to the HC, cancel all the URBs we had submitted and not yet done and when all that is done, the xfer callback will be called--this will call the URB callback.

Glossary

DWA -- Device Wire Adapter

USB host, wired for downstream devices, upstream connects wirelessly with Wireless USB.

EVENT -- Response to a command on the NEEP

HWA -- Host Wire Adapter / USB dongle for UWB and Wireless USB

NEH -- Notification/Event Handle

Handle/file descriptor for receiving notifications or events. The WA code requires you to get one of this to listen for notifications or events on the NEEP.

NEEP -- Notification/Event EndPoint

Stuff related to the management of the first endpoint of a HWA USB dongle that is used to deliver an stream of events and notifications to the host.

NOTIFICATION -- Message coming in the NEEP as response to something.

RC -- Radio Control

Design-overview.txt-1.8 (last edited 2006-11-04 12:22:24 by InakyPerezGonzalez)