My bandwidth is wider than yours

Ultra Wideband, Wireless USB, and WiNET in Linux*

Iñaky Pérez-González
Open source Technology Center, Intel Corporation
`inaky.perez-gonzalez@intel.com`

Abstract

Imagine a radio technology that gives you 480 Mbps at short range. Imagine that you don't have to keep all those cables around to connect your gadgets. Make it low-power, too. You even want to be able to stream content from your super-duper cell phone to your way-too-many inches flat TV. Top it off, make it an open standard.

It's not just a dream: we have it and is called Ultra-Wide-Band, and it comes with many toppings of your choice (Wireless USB and WiNET; Bluetooth and 1394 following) to help remove almost every data cable that makes your day miserable. And Linux already supports it.

1 What is all this?

UWB is a high speed, short range radio technology intended to be the common backbone for higher level protocols. It aims to replace most of the data cables in desktop systems, home theaters, and other kinds of PAN-like interconnects. It has been defined by the WiMedia consortium after a long fight in IEEE over the underlying implementation and now it is ECMA-368 and is back in IEEE for further standardization.

It provides all the blocks (delivery of payloads, neighborhood and bandwidth management, encryption support, {broad,multi,uni}cast, etc.) needed by higher level protocols to build on top without any central infrastructure.

Wireless USB sits on top of UWB, where it allocates bandwidth and establishes a *virtual cable*; WUSB devices connect to the host in the same master-slave fashion as wired USB. The security of the cable is replaced with strong encryption and an authentication process to rule out snooping and man-in-the-middle attacks. Backwards compatibility is maintained, so we can reuse all of our drivers with slight modifications in the core host stack.

WiNET snaps an Ethernet frame over a UWB payload and adds a few protocols for bridging; devices cluster in different WiNET networks (similar to WIFI ad-hoc) and also includes authentication and strong encryption.

Other high level protocols can build on top of UWB (Bluetooth 3.0 is planning to do so, for example).

2 Ultra Wide Band

This protocol is designed for being low-power (as in little usage and efficient), with good facilities for QoS and streaming (mainly centered in audio and video) and providing strong cryptography on the transport to compensate for the open medium.

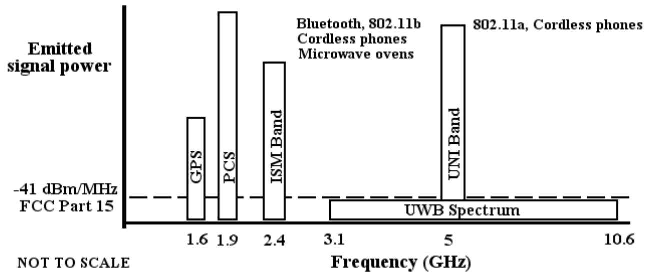

Figure 1: UWB's spectrum usage

Ultra-Wide-Band operates over the unlicensed 3.1 to 10.6 GHz band, transferring at data rates from 53Mbps to 480Mbps;[1] high rates reach up to 3 meters; lower rates, all the way to 10 meters. These are split in fourteen 528 Mhz bands; these are grouped in five band

[1]53.3, 80, 106.7, 160, 200, 320, 400, and 480 Mbps, not all mandatory.

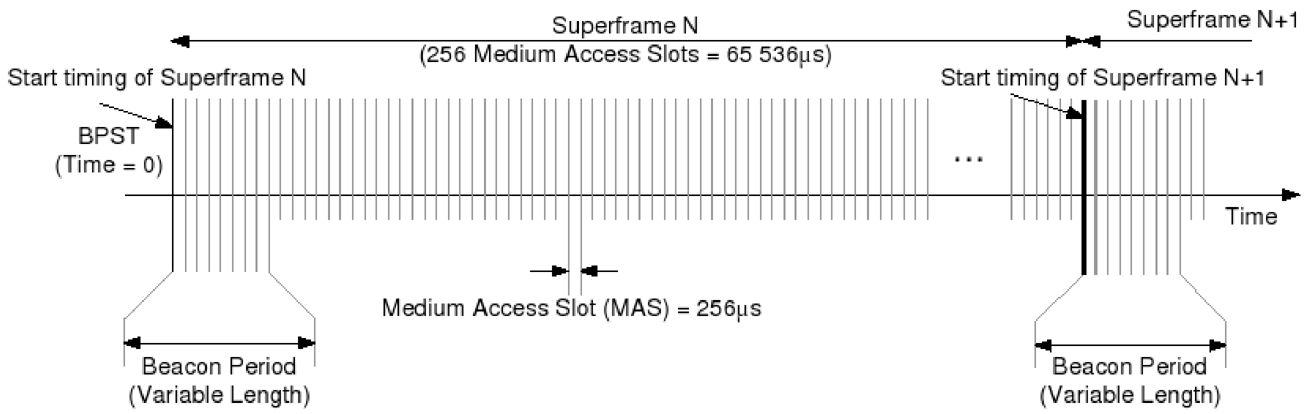
Superframe N
(256 Medium Access Slots = 65 536μs)
Superframe N+1
Start timing of Superframe N
Start timing of Superframe N+1
BPST
(Time = 0)
...
Time
Medium Access Slot (MAS) = 256μs
Beacon Period
(Variable Length)
Beacon Period
(Variable Length)

Figure 2: Division of time in UWB (credit: ECMA-368 Fig 3)

groups (*channels*) (composed of three bands each except the last one, which are two). Data is encoded MB-OFDM[2] over 122 sub-carriers (100 data, 10 guard, 12 pilot).

The power emission is as low as the maximum specified in the FCC Part 15 limit for interference: $-41 dBm/Mhz$ ($0.074 \mu W/MHz$), being the practical radiated power about $100 \mu W/band$ ($-10 dBm$). This is more or less three thousand times less than a cell phone[3] and allows UWB to appear as noise to other devices.

Time is divided in *superframes* (see Figure 2), composed of 256 *media allocation slots* (MAS), $256 \mu s$ each. Thus a *superframe* is about $65 ms$. The *MAS* is the basic bandwidth allocation unit.

The *superframe* starts with the *beacon period*, which is divided in $85 \mu s$ *beacon slots* (96 maximum, about 32 MAS slots). The first beacon slots are used for *signalling*; when a new device wants to join it senses for it to be empty and transmits its beacon until it is assigned another empty slot.

It is important to note that devices don't have a common concept of *start of the superframe*. There might be an offset and devices coordinate among themselves to synchronize in a common start of superframe. That is called a *beacon group*, a group of devices that beacon during a shared beacon period at the beginning of the same superframe. This becomes a complication for a device B that can hear beacons from A and C without A and C being able to listen to each other.

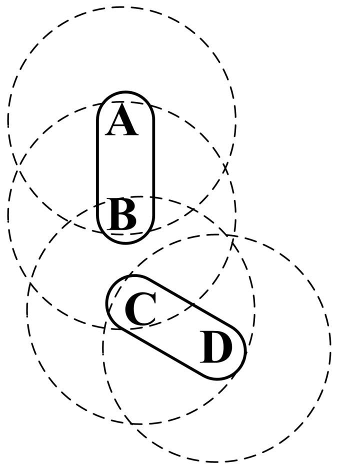

Figure 3: Hidden neighbours

If A and B are beaconing in the same beacon group and C gets in range of B (Figure 3), C might be beaconing at a time different to that of A and B.[4] B recognizes C's beacon as an *alien beacon* and tells A about which slots C is using; thus A and B don't try to use those slots to transmit (as their tranmission would get mixed with C's).

The rules for use of the media are extremely simple. A device might only transmit:

- Its beacon, during the beacon slot assigned to it.

[2]Multiband Orthogonal Frequency Division Modulation.

[3]Roughly calculated, about five orders of magnitude less than WIFI.

[4]Especially if it has a beacon group formed with device D, for example.

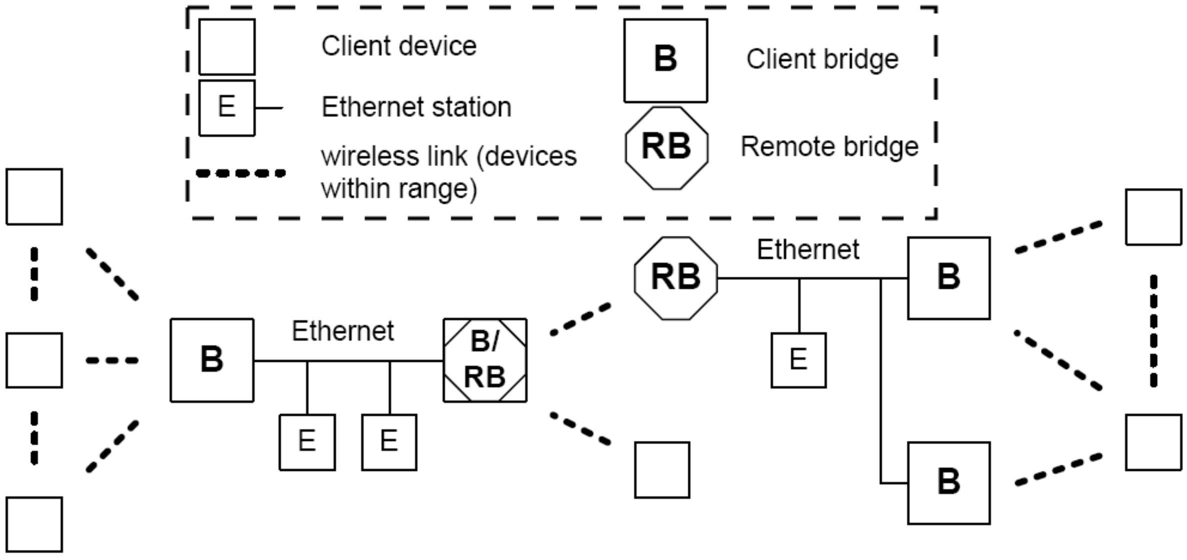

Figure 4: Full WiNET topology

- Data, during the MAS slots reserved by it. This is called *Distributed Reservation Protocol* (DRP), basically a TDMA model. It involves a negotiation among the devices on which devices own which MAS slots. Each device can define a maximum of eight static *streams* (or allocations) with this method. They are fixed (as in reserved bandwidth) until dropped.
- Data, when the media is not being used. Called *Prioritized Contention Access*, it is a CSMA technology: sense the carrier and if empty, transmit. Eight different priority levels are defined for which devices contend based on the prioritization they give to their data.

UWB aims to provide a secure enough media, tamper- and snoop-proof. It is implemented using AES-128/CCM, one-time pads, and 4-way handshakes for devising pair-wise and group-wise temporal keys (secret establishment/authentication to avoid man-in-the-middle attacks is left to the higher level protocols).

Consideration is given to power saving. Devices can switch off their radio until they have to transmit/receive (beacon or data), even advertising to each other that they are suspending beaconing for an amount of time. Transmission rates and emission power can be adjusted to decrease consumption of energy, for example, for devices in close proximity.

In general, UWB becomes a flexible low-level protocol for building on top. It offers enough flexibility for all kinds of media and data, and almost no restrictions in mobility (other than range) and usage models.

3 WiNET: IP over UWB

WiNET slaps an Ethernet payload over UWB frame to achieve the same functionality that is possible with Ethernet: IP, bridging, etc.

In concept, it is very similar to WIFI, except for the short range (10m max; a brick wall will stop it short[5]) and the nonexistence of access points (all is ad-hoc). It is faster and more power-efficient for PAN usage models.

WiNET-capable devices group in *WiNET Service Sets* (WSSs).[6] Devices may belong to more than one at the same time.

Security is provided using UWB's framework,[7] which provides data integrity and privacy; there are association methods to avoid man-in-the-middle attacks when establishing a trust relationship (using numeric comparison or simple password comparison).

[5]Which is an advantage in many usage models

[6]Roughly equivalent to the *ESSID* in WIFI terms.

[7]AES-128/CCM, 4-way handshakes to generate pair and group wise keys, one time pads.

QoS is achieved by reserving fixed point-to-point streams allocated with the UWB Dynamic Reservation Protocol (for example, the bandwidth required to stream audio to the living room speakers is known ahead of time) or by mapping IP traffic prioritization into the UWB Prioritized Channel Access traffic levels.

As well, 802.1D bridging services are provided (see Figure 4) to allow different WiNET segments to be bridged. This is useful, for example, for providing wireless network connectivity in high density urban dwellings, where the high number of apartments would make a WIFI access point on each unfeasible. Under this model, an Ethernet backbone connects WiNET *access points* that mobile devices connect to while roaming through the apartment without losing connectivity.

4 Wireless USB

Wireless USB by itself is very simple: remove the cable from USB 2.0, put in its place an UWB radio. The rest (at the high level) remains the same; with a few modifications to the USB core stack we can (in theory) reuse most of our already written drivers for mass storage, video, audio, etc.

WUSB still maintains the master/slave model of the wired version (even if UWB is peer to peer). The WUSB host creates a static bandwidth allocation with the Distributed Reservation Protocol, and in there it creates a *WUSB Channel*. Devices that connect to that channel define (along with the host) a *WUSB Cluster*.

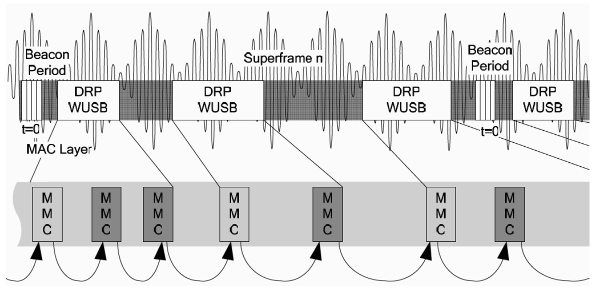

Figure 5: A WUSB Channel (credit: WUSB1.0 Fig 4-4)

At the beginning of each allocated period of time, the host emits an MMC (*Microscheduled Management Command*). This is a data structure composed of *information elements* (IEs) that specifies the length of the allocated period, which devices can transmit and when, gives time for devices to query the host (*device notifications*, DNs[8]), and provides a link to the next MMC, which prefixes another allocation period.

This model allows WUSB devices to be simplified, as they don't have to understand (unless desired) UWB, beaconing, DRP, or PCA. They just look for MMCs and follow the links, getting their I/O control information from them.

Cable-based concepts, such as reset, connect, and disconnect are implemented via signalling in the *device notification* time slots (for device to host) and in the MMC information elements (host to device).

4.1 Wireless USB security

WUSB security builds on top of UWB's security framework.[9]

The trust relationships are established via a *Connection Context*, which is composed of a *Connection Host ID* (CHID), *Connection Device ID* (CDID), and *Connection Key* (CK). The CHID uniquely identifies the host, the CDID the device, and the CK is the shared secret. Both host and device keep the same CC as proof of trust.

Establishing the secure connection is done via the 4-way handshake process. When a device is directed by the user to connect to the host, it looks for the CHID broadcasted by it, and then looks up in its internal CC tables for the CDID it was assigned by that host. Then host and device prove to each other that they have the CK without actually exchanging it and derive a pair-wise temporal key. The host issues a new group key and issues it to all devices, including the new one. As well, all keys expire after a certain number of messages have been encoded with them, time at which new ones are renegotiated using another 4-way handshake.

When there is no Connection Context established, WUSB specifies methods for the authentication/pairing process:

- **Cable Based Association:** For devices that can connect with a cable, it is used to establish trust by transmitting the connection context using a *Cable-Based Association Framework* [USB] interface.

[8]Unlike wired USB, in WUSB devices can initiate transactions to the host.

[9]AES-128/CCM, 4-way handshakes, pair and group wise keys, one time pads.

- **Numeric Association:** Use Diffie-Hellman to create a temporary secure channel and avoid man-in-the middle attacks by having the device and the host present a short (two to four digits) number to the user. If they match, the user confirms the pairing. Limited range and explicit user conditioning make up for the lack of strength in a four-digit decimal hash.

Devices can keep more than one Connection Context in non-volatile memory, so that there is no need for re-authenticating when moving a device from one host to another.

5 The hardware

Hardware comes in the shape of a UWB Radio Controller (*RC*) with support on top for the higher level protocols. There is specialization, however: a low-power device might sacrifice some functionality to save power; a PC-side controller would offer full support.

We will concentrate mostly on the host side, as we just want to use the devices.

5.1 USB dongles: HWAs or Host Wire Adapters

Defined by the Wireless USB specification, HWAs consist of a UWB radio controller and a WUSB host controller all connected via USB to the host system. Other extra interfaces are possible (for example, for WiNET).

WUSB traffic to/from WUSB devices is piped through wired USB. Imagine a USB controller connected via USB instead of PCI.

Its main intent is to enable legacy systems to get seamless UWB and WUSB connectivity. The drawback is the high overhead of piping USB traffic over USB.

5.2 Wireless USB hubs: DWAs or Device Wire Adapters

Defined by the Wireless USB specification, a DWA is composed of a hub for USB wired devices whose upstream connection is wireless USB. Similarly to the HWA, think of a USB host controller that is connected to the system via *Wireless USB*, not PCI.

This is intended to connect your wired devices to your Wireless-USB-enabled laptop; as well, legacy applications will use much of this. Take an existing USB chipset for some kind of device, put in front a DWA adapter, and suddenly your device is “wireless.”[10]

It has the same drawbacks as HWA regarding performance.

5.3 PCI (and friends) connected adapters: WHCI

Defined by the Wireless Host Controller Interface, the brother of EHCI. It is a Wireless USB host controller plugged straight to the PCI bus, which, as HWA, might contain other interfaces (again, such as a WiNET interface).

This is intended for new systems or those where a PCI or mini-PCI card can be deployed easily. It gives the best performance, as there is no extra overhead for delivering the final data to its destination (as in HWA/DWA).

Exercise for the reader: what happens when you connect a WUSB printer to a HWA, the HWA to a DWA, and the DWA to a WHCI, finally to your PC?

6 Linux support

Full-featured Linux support for UWB, Wireless USB, and WiNET will consist of:

- A UWB Stack to provide radio neighborhood and bandwidth management.
- Drivers for HWA (USB dongle) and WHCI (PCI) UWB Radio Controllers plugging into the UWB stack.
- A Wireless USB stack providing two abstractions used by the three different kinds of host controllers (Wireless USB Host Controller and Wire Adapter). It also includes security, authorization/ pairing management, and seamless integration into the main USB stack.
- Drivers for HWA (USB dongle) and WHCI (PCI) Wireless USB host controllers, as well as for the DWA (Wireless USB hub) host controller.

[10]And yes, if you connect twenty of these, you’ll have twenty USB hosts in your machine.

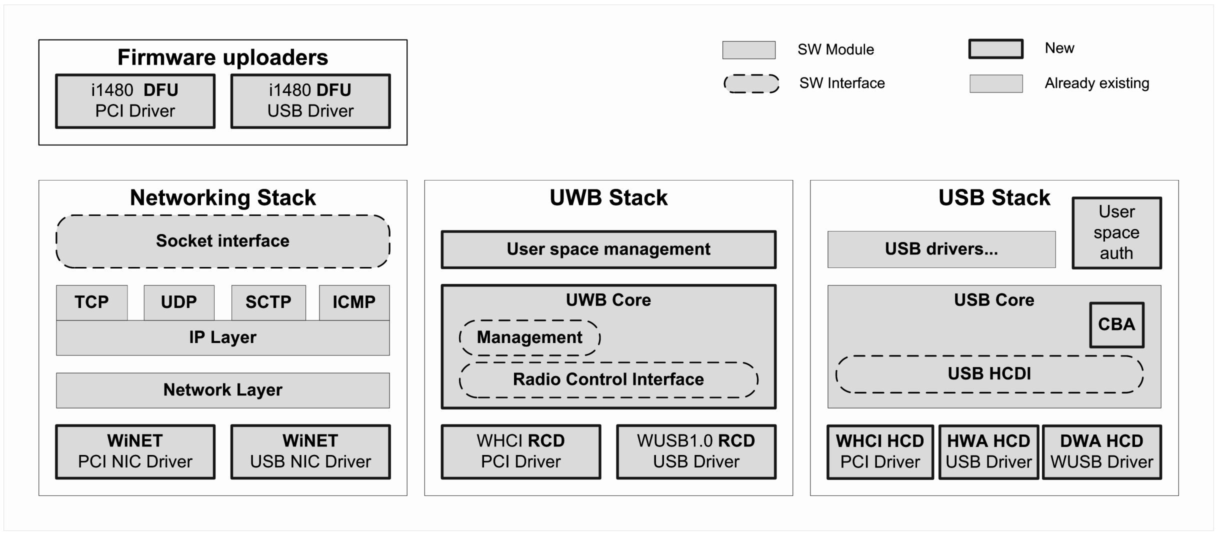

Figure 6: Linux's support for UWB/WUSB/WiNET

- Driver for WUSB Cable-Based Association, as well as support for WUSB numeric association.
- WiNET drivers: Our efforts are also to implement, using the Intel® Wireless UWB Link 1480, network device drivers for the USB and PCI form factors.

This adds up to two stacks and six drivers; eight, if we count the WiNET drivers (Figure 6).

6.1 Current status

In general, the driver set is quite stable and usable with the available hardware (which is very scarce).

The Linux UWB stack has implemented most of the basic management features, allowing the discovery of remote devices and a Distributed Reservation Protocol bandwidth negotiator. It can work with devices implementing WUSB 1.0 and WHCI 0.95.[11] Radio control drivers have been implemented for HWA (`hwa-rc`) and WHCI (`whc-rc`).

As of April 2007, only the HWA Wireless USB host controller is implemented, in a limited fashion; only control, bulk, and interrupt transfers work. However, it is possible to use an AL-4500 mass storage evaluation device made by Alereon. Authorization/pairing is being implemented, making the low-level Cable-Based Association driver complete (user space glue is now done manually).

We have also developed drivers for the Intel 1480 Wireless UWB Link WiNET interface, which are quite stable as of now.

In general, there is still a lot of work to be done. The UWB stack still needs to handle a lot of complex situations, like alien beacons, suspend and resume handling in the UWB media, selection of ideal rate and power transmission parameters, and fine tuning of the bandwidth allocator. The Wireless USB stack requires a complete transfer scheduler and implementation of isochronous support. The driver for DWA needs to be put together (with pieces from HWA) and a driver for the WHCI WUSB host controller has to be created.

6.2 Sample usage scenarios

All interaction between the user and the local radio controllers happens through sysfs:

```
# cd /sys
# ls -N bus/uwb/devices/
uwb0/
# ls -N class/uwb_rc/
uwb0/
```

[11]Both specs are mostly identical; however, as WHCI is developing on aspects that were not yet known when WUSB 1.0 was finalized, errata will be issued to correct them.

The radio is kept off by default; we could start beaconing to announce ourselves to others in one of the supported channels[12] on station A:

```
A# cd /sys/class/uwb_rc
A# echo 13 0 > uwb0/beacon
```

If now a user starts scanning on station B:

```
B# echo 13 0 > uwb0/scan
```

It will find station A's beacon and will be announced by the kernel; entries will be created in `sysfs`:

```
B# tail /var/log/kern.log
...
uwb-rc uwb0: uwb device \
(mac 00:14:a5:cb:6f:54 dev f8:5c)\
 connected to usb 1-4.4:1.0
...
B# ls -N /sys/bus/uwb/devices/
uwb0/
f8:5c/
```

This indicates that a remote UWB device with address `f8:5c` has been detected. At this point, the devices are not *connected*, but just B is listening for A's beacon. For them to be able to exchange information, B needs to also beacon so A knows about it—they need to be linked in the same *beacon group*. That we accomplish by asking B to beacon against A's beacon (in the same beacon period):

```
B# echo f8:5c > uwb0/beacon
```

Now we'll see in station B a similar message (`uwb device (...) connected`) as well as an entry with its 16-bit address in `/sys/bus/uwb/devices`. In the case of the Intel 1480 Wireless UWB Link (USB form factor), we could now load the WiNET driver (`i1480u-winet`) and configure network connections on both sides:

```
# modprobe i1480u-winet
# ifconfig winet0 192.168.2.1
```

[12]Most hardware known to our team supports channels 13, 14, and 15.

Connecting Wireless USB devices

If we had any Wireless USB devices, we would have to tell the WUSB controller to create a WUSB Channel; assuming controller `usb6` is the one corresponding to our radio controller:

```
# cd /sys/class/usb_host/usb_host6
# echo <16 byte CHID>   \
       0001             \
       mylinux-wusb-01  \
  > wusb_chid
```

With this we have given a 16-byte CHID to the driver, which has configured it into the host so it broadcasts a WUSB channel named `mylinux-wusb-01`. Now devices that have been paired with this CHID before will request connection to the host when we ask them to (for now we allow all of them to connect):

```
new variable speed Wireless USB \
device using hwa-hc and address 2
```

From now on, this behaves like yet another USB device. There are still no controls for selecting rate or power.

If no transactions are done to the device or the device doesn't ping back to the host's keep-alives, then it will be disconnected. The timeout is specified in a per-host basis in file `/sys/class/usb_host/X/wusb_trust_timeout`. Most devices we've seen until now don't implement keep-alives, so this value has to be set high to avoid their disconnection.

7 Conclusion

We have done a quick description of Ultra Wideband, Wireless USB, and WiNET, a set of technologies that aim at replacing all the cables that clutter desktops and living rooms. Designed with streaming and power efficiency in mind, they also provide security comparable to that of the cable.

We have also described how Linux implements support for it, its current status, and how to use it. There is basic support working, but a lot is still to be done.

We expect a huge increase of consumer electronics devices of all kinds (cellphones, cameras, computing, audio, video...) supporting this set of technologies in upcoming years. And with Linux playing an all-the-time more important role in those embedded applications, as well as in the desktop, it is key that it supports them as soon as they hit the streets en masse.

References

[linuxuwb.org] Linux UWB/WUSB/WiNET, `http://linuxuwb.org`

[WiMedia] WiMedia Alliance, `http://wimedia.org`

[ECMA368] Ecma International, *ECMA-368 High Rate Ultra Wideband PHY and MAC standard, 1.0*, `http://www.ecma-international.org`

[WiMedia] WiMedia Alliance, *WiNET specification* (still not publicly available) `http://wimedia.org`

[WUSB1.0] USB Implementors Forum, *Wireless USB 1.0 specification*, `http://usb.org`

[WAM1.0] USB Implementors Forum, *Association Models supplement to the Certified Wireless Universal Serial Bus Specification, 1.0*, `http://usb.org`

[WHCI] Intel Corporation, *Wireless Host Controller Interface, 0.95*, `http://intel.com`

Proceedings of the Linux Symposium

Volume Two

June 27th–30th, 2007
Ottawa, Ontario
Canada