Windows Rally™ Development Kit

LLTD Porting Kit User's Guide

November 1, 2006

Abstract

This document accompanies the sample code that is provided in the Link Layer Topology Discovery Porting Kit (LLTD Porting Kit). It describes the code design and implementation details so that implementers of the LLTD protocol can incorporate it in their products that use Internet Protocol (IP) connections, thereby enhancing their product's interoperability with Microsoft® Windows® operating systems.

LLTD is a key component of the Microsoft Windows Rally™ set of technologies.

The current version of this paper is maintained with the Porting Kit on the Web at:
http://www.microsoft.com/whdc/rally/rallykit.mspx

Contents

Disclaimer

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Introduction to the LLTD Porting Kit

This document accompanies the sample code that is provided in the Link Layer Topology Discovery Porting Kit (LLTD Porting Kit). It describes the design and implementation details of the sample code so that implementers of local area networking devices can incorporate the LLTD protocol in their products, thereby enhancing their product's interoperability with Microsoft® Windows® operating systems.

The LLTD protocol is designed for a local area network (LAN). It cannot and should not be routed; when implemented in routers or edge devices, LLTD should not be running on the device's WAN port. The protocol serves three primary purposes:

- PC and device discovery
- Topology inference
- Network diagnostics

LLTD is, as its name suggests, a mapping and diagnostic protocol that runs entirely on layer-2 (link-layer) network connections such as Ethernet and 802.11 wireless.

Formerly, the protocol was referred to as LLD2, but this term has been replaced with "LLTD QoS Extensions." LLTD and LLD2 should be treated synonymously.

The sample implementation of the LLTD responder in the porting kit can compile and run (as a user-mode daemon) on almost any recent Linux-based (2.4-kernel) computer or embedded Linux device (such as the Cisco/LinkSys WRT54GS or Buffalo Airstation). The LLTD responder provides the functionality that is necessary to actively participate in the Windows Network Map/Network Explorer features in the Windows Vista™ operating system and to participate in bandwidth estimation and the Quality of Service (QoS) experiments that are necessary to reliably stream media in home networks.

The kit provides an easy-to-follow working code example from which a manufacturer of any Internet device can develop a protocol stack that allows the device to appear in Windows Network Map or Network Explorer and actively participate in the mapping process.

Important: This guide assumes the reader is familiar with the "LLTD: Link Layer Topology Discovery Protocol" specification, which is part of the Microsoft Windows Rally™ set of technologies. The specification is available from the Rally Web site at:
http://www.microsoft.com/whdc/rally

LLTD Porting Kit File List

File	Description
band.c	BAND (Block-Adjust Neighbour Discovery): network load limiter
band.h	BAND data structure
bandfuncs.h	Prototypes for BAND
common.mk	Common Makefile build rules
ctmain.c	Confidence Test (CT) main entry point
ctosl-linux.c	
ctpacketio.c	
ctstate.c	
enumeration.c	Compilation module for smE (enumeration process state machine)
enumeration.h	Protocol specifics for enumeration process
event.c	Central **select()** loop unifying IO events with timed events
event.h	Prototypes for events

File	Description
globals.h	Global variables
lld2d.conf	Configuration file specifying the icon file name
lld2_types.h	Protocol specifics for access types
main.c	**Main()** and command-line option parsing
Makefile	Build rules
mapping.c	Compilation module for smT (topology mapping state machine)
osl.h	Portable OS Layer (OSL) API
osl-linux.c	Linux implementation of OSL API
packetio.c	Packet receive handler, validation, packet formatting, and transmission
packetio.h	Prototypes
protocol.h	Ortelius frame formats and other protocol specifics
qosglobals.h	Globals for QoS Extensions for LLTD
qospktio.h	Packet IO for QoS Extensions for LLTD
qosprotocol.h	Protocol specifics for QoS Extensions for LLTD
S75lld2	
seeslist.c	Fixed size circular queue holding src/dst of observed packets
seeslist.h	Prototypes
session.h	Per-interface protocol state
sessionmgr.c	Compilation module for smS (session status state machine)
smevent.h	Protocol specifics for state machine events
state.c	Main protocol state machine
statemachines.h	Prototypes
tlv.c	Type-Length-Value (TLV) formatting routines
tlv.h	Prototypes
tlvdef.h	Definition of supported TLVs
tux.ico	Icon file for x86 Linux PC
util.c	Miscellaneous utilities
util.h	Prototypes
wrt54g.large.ico	Icon file for WRT54G
wrt54g.small.ico	Icon file for WRT54G

LLTD Topology Discovery and Mapping Techniques

LLTD is a sophisticated protocol that can and should be incorporated into any IP-connected device, to provide device discovery, problem resolution, and network diagnostics functions. Its primary purpose is to facilitate creation of a map of the network to which a particular Windows PC (called the *mapper*) is connected and then explore the capabilities and capacities of that LAN with probes and diagnostics. The LLTD protocol is an integral part of every Windows Vista PC, providing Layer-2 discovery, topology mapping, and bandwidth estimation.

The Mapper: Discovery Packets

To maximize its usefulness in a new, unconfigured network, the mapper functionality runs at the link layer (layer 2 of the 7-layer OSI Networking model) and so does not require or use Transmission Control Protocol/Internet Protocol (TCP/IP) data units. This is done deliberately to aid in diagnosing problems at the IP (and higher) layers, and to help identify connectivity problems or poor network topologies.

To make its maps, the mapper broadcasts Discovery packets and listens between the transmissions for devices—the *responders*—which broadcast back to it an

announcement of their existence. These announcements are called *Hello packets*, and the protocol specifies a new algorithm for spreading the responses out in time to avoid scaling problems on LANs that contain more than a few responders. The block adjust node discovery (BAND) algorithm attempts to make the mapping process robust in a large-scale environment without making it too slow in a small-scale environment, by having each responder watch all of the Hello packet traffic and varying a back-off delay for its own Hello packet according to how many others have already responded.

The Responder: Hello Packets

In normal use, the process starts when a mapper transmits its first Discover packet and the responders who hear it randomly disperse their Hello responses. The mapper waits only a short time and then sends another Discover packet and resumes listening. Because of scaling considerations, not all responders answer in the first listening period or in any particular listening period. The mapper, therefore, must keep sending Discover packets until it is convinced that no further responders remain to be revealed, a mere few seconds in a small environment.

To reduce the conflict of Hello packets in a larger environment, responders must respond only until the mapper "acknowledges" the responders, in the body of a Discover packet that is received subsequent to their Hello packet transmission. Then the responder is said to be "associated" with the mapper in a mapping session and no further Hello responses are required unless another simultaneous session starts up.

Net Metadata Mining: TLVs

Hello packets carry more than just a presence announcement. Important pieces of data such as the responder's name, its function, its physical medium attachment, its link speed, and a representative icon are returned in the body of the Hello packet. These pieces are sent as a variable-count series of triples called type-length-value (TLVs). These are variable-length structures that consist of 1 byte of type, followed by 1 byte of length, and followed (usually) by <length> bytes of value. The LLTD: Link Layer Topology Discovery Protocol specification currently defines the type codes for about 20 pieces of data and one final type code (Type=0), which terminates the variable-count series of TLVs in the Hello packet.

Neighbor Visibility: Emits, Probes, and Trains

After the mapper receives all of the Hello packets that it expects, it sends targeted packets (called *Emits*) to each responder, asking them to transmit special mapping packets that are called *Probe packets* or *Train packets,* by using MAC-level addresses that the mapper chose. Train packets inform switches of the existence of a given address. This allows the mapper to distinguish switches from hubs in the connectivity topology because untrained switches flood a packet with a new address to all ports but trim nonessential ports after the destination's port is known.

The mapper-chosen addresses that are used in these Train and Probe packets are selected from a group that is allocated to Microsoft so as not to conflict with the hardware that is being mapped. Emit commands may ask the responder to emit several successive Train and Probe packets, with an individually specified delay preceding each of them. After the responder sends all of its "emitees" out on the wire (or over the air), it sends a brief acknowledgment to the mapper that indicates completion.

The purpose of these emitees (Probe and Train packets) is to establish which of the responders can see each other and under what circumstances (such as sharing a

hub or connected to different ports of a switch). In turn, each responder is asked to perform an Emit command, and then the mapper queries each of the other responders to see if they heard the emitting-responder's Probe responses. Responders report their *sees-list* (the list of contrived MAC addresses that was saved from packets that they have seen since the last query) in a packet that is called a QueryResponse and then discard the sees-list in preparation for the next set of Probe packets. This requires a small amount of storage on the LLTD-compliant responder to maintain the sees-list during the brief mapping session.

With all this information about who can see whom, the mapper now draws a connectivity map of the responders, including devices that do not respond like switches and hubs, with some degree of accuracy.

Metadata Revisited: Small and Large TLVs

In large environments, even the small amount of data that the TLVs return might be onerous and cause collisions, so the protocol allows the Hello packet to essentially say, "I have this data type, but you must ask for it specifically if you want it." These LargeTLVs (or LTLVs) are also used to publish information that is too substantial to fit in the relatively tiny maximal Ethernet packet (1,400 to 1,500 bytes). One particular example is a TLV that is normally reported as an LTLV is the device's icon file. The responder can provide the mapper with a Windows icon that represents it in a Windows Vista map. Of course, even a minimal colored icon of that size requires 2,000 bytes or more and so is returned as a LTLV in the Hello packet.

Subsequent to the mapping, the mapper can ask its responders individually to provide the LargeTLVs in which it is interested. The mapper continues to ask for a particular type's LargeTLV, supplying an offset into that data for the responder to start its report until the responder replies with a final block of data marked that is "No More Data". For example, in this way large icons can be accommodated through several packet exchanges. The demonstration code provides full support for easily defining TLVs and returning them in either small or large formats.

Scalability, Diagnostics, and More

The protocol has several more components that are used to prevent hijacking, short-circuit combinatorial explosion, and handle dropped packets, among other things. Another section of the protocol (not covered in this document) is concerned with diagnostics and performance measurements.

Implementing the LLTD Daemon

This kit provides a reference implementation of the LLTD protocol, written in C for POSIX-compliant operating systems. The goal for this implementation is to provide a fully featured reference implementation that is easily ported for other systems.

Building the LLTD Daemon

Support for different UNIX flavors is achieved by selecting an appropriate OS Layer (OSL). Edit the Makefile to change OS_LAYER to pick an appropriate *osl-FOO.c* file for your operating system. If there is no suitable OSL and you want to write your own, see "Porting to Other Environments" later in this document.

You will need GNU make to run the Makefile because it uses the *$(patsubst)* extension to calculate the list of *.o* files to build. If you do not have a GNU make, you can manually edit the OBJFILES definition to explicitly list all of the required *.o* files.

Summary steps to build the LLTD Daemon

1. Edit Makefile to select the OS_LAYER.
2. Run **$ make.**
3. Complete a test run:

   $ su
   # ./lltdd -d eth0 (or whatever the appropriate interface name is)

   Expect output similar to the following:

   ```
   finished mapping: going quiescent
   ENTER: Quiescent
   lltdd: listening on interface 00:50:04:4d:19:23
   ^C
   ```

4. Install the LLTD Daemon binary in an appropriate place, and then arrange for the LLTD Daemon to be run as root when an interface is brought up.

Setting Up and Using the Porting Kit

This section describes how to implement the LLTD Porting Kit on a sample embedded device (Wi-Fi router).

Assumptions:

- You have a PC with Windows Vista installed (the *mapper)*.

  This PC is cabled to the responder (as shown in Figure 1). If it has a second Ethernet interface, connect that interface for Internet access.

- You have a PC with a POSIX-compliant operating system that can be used to compile the embedded run time and (if desired) to respond to mapping requests itself. This is the *devbox*.

  Notes:

  - You might want to run a packet sniffer, such as NetMon or Ethereal, on the devbox to see the protocol exchange occur between the Responder and the Windows Vista PC.
  - Capturing the protocol exchange between two Windows Vista PCs is a helpful way to see the proper implementation of the protocol.
  - The Windows Vista PC must be set to a *Private* network profile for mapping and LLTD responses to be enabled. LLTD is disabled if a network adapter is set to a *public* profile. This is to prevent spurious mapping of PC's in public environments such as public Wi-Fi hotspots.

- You have a MIPS-based wireless router device, such as a WRT54GS or a Buffalo Airstation WBR2-G54S (the *responder*).
- You have cabled the Windows Vista mapper PC and the devbox to a hub and connected the hub to the first (leftmost) LAN port on the responder.

  You can omit the hub if you want to, but it is useful to allow the devbox to watch the wire for packets that are coming from all sides, rather than just depend upon the Windows Vista PC to capture things that might not be coming through the switch.

Figure 1 illustrates this configuration:

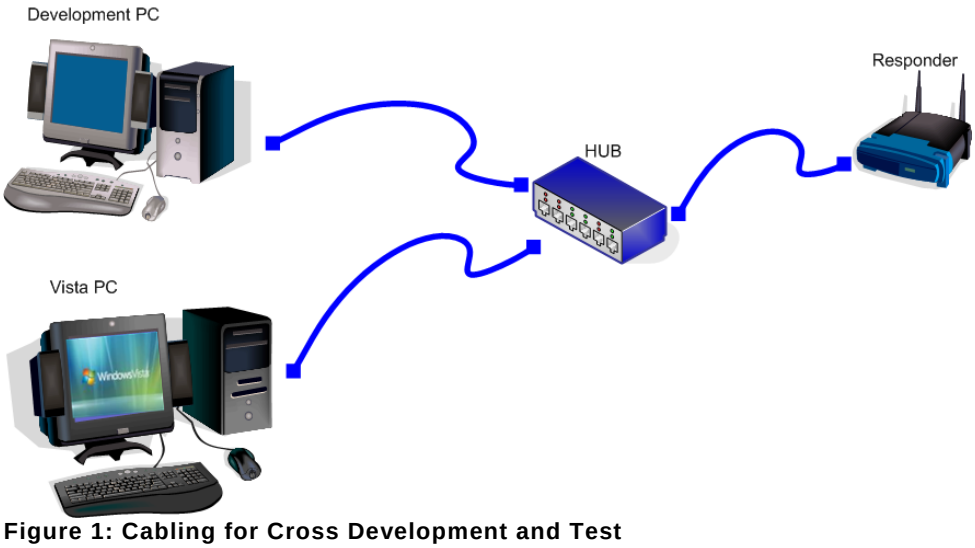


Figure 1: Cabling for Cross Development and Test

Using the LLTD Daemon

Run the LLTD Daemon in debugging mode to see all the debug tracing that it does.

To run the LLTD Daemon

1. Run the following from the command line:

   /sbin/lld2d -d -t 31 br0

   Or, if you have built the x86 version and are in its build directory, run the following from the devbox command line:

   ./lld2d -d -t 31 br0

2. Try them both at the same time with the mapper, where:

   - **-d** means "debugging—don't daemonize, so messages come to the console."
   - **-t 31** means "set the trace flags to 31 (0x1F)."

   The trace flags from *Globals.h* are as follows:

   ```
   TRC_BAND    = 0x1
   TRC_PACKET  = 0x2
   TRC_CHARGE = 0x4
   TRC_TLVINFO = 0x8
   TRC_STATE   = 0x10
   ```

   TRC_ALL == 0x1F or decimal 31 and br0 is the interface that you want to serve Bridge-0, which combines eth1 and wl0.

Before making a map, ensure that your Windows Vista PC has connectivity with the responder.

To check the Windows PC connectivity with the responder

Type the following at the command line, if you are using WRT as the responder:

ping 192.168.1.1

If you are using the devbox or a Buffalo, for example, substitute the appropriate IP address. The responder replies to each ping if connectivity is properly wired.

To make a map in Windows Vista

Click **Start > Network** > **Network and Sharing Center** > **View Full Map**.

LLTD Daemon Architecture

Initially, the LLTD Daemon runs as root, allowing it to open a raw Ethernet socket to receive LLTD packets (ethertype 0x88D9). The LLTD Daemon then drops root privileges and goes into a **select()** loop, listening for packets on the socket. This is termed the Quiescent state.

The LLTD Daemon wakes from Quiescent upon receiving a MapBegin protocol packet from a mapper application. The daemon enables promiscuous mode on the interface and starts recording the source and destination Ethernet addresses for Probe-type protocol packets and for ARP Response packets to specific addresses.

When topology discovery is over, the mapper sends a Reset protocol packet, which causes the LLTD Daemon to move back into the Quiescent state and disable promiscuous mode. Quiescent is also entered upon timeout if the LLTD Daemon does not hear from the mapper for a while (approximately 60 seconds).

The LLTD protocol is Ethernet-layer only, so it will propagate only within a single Ethernet broadcast domain, typically a single VLAN or IP subnet.

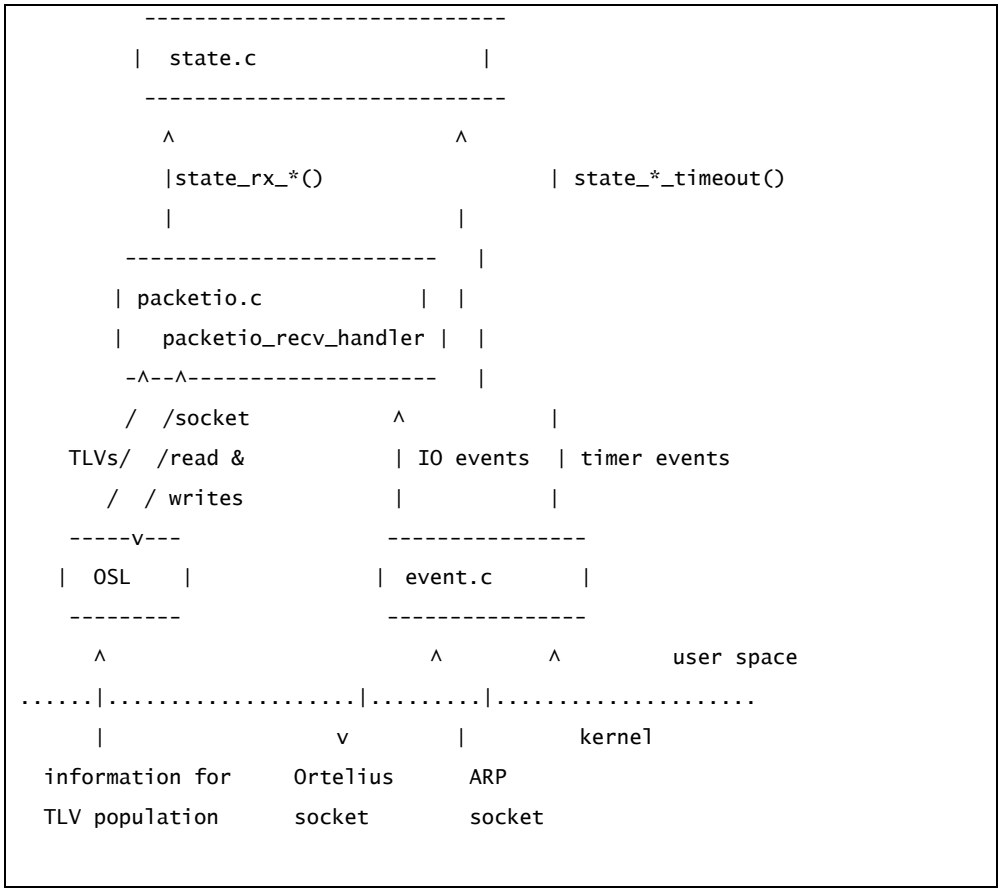


Figure 2. Schema of LLTD Daemon Subsystems

OSL and *event.c* are the main points of contact with the underlying operating system. The file *packetio.c* handles reception of packets, filtering out uninteresting packets (in case the kernel cannot), parsing and validating them for correctness, retransmitting lost responses to requests, and passing protocol message receipt indications—**state_rx_*()**—to the higher-level protocol state machine in *state.c*.

State.c clocks its state machine according to these indications, possibly updating its session state, and calling back down to *packetio.c* to transmit a response if required. *State.c* can also call into *event.c* to set up a timer event—that is, **state_*_timeout()** function—to be called when a certain time passes. These timeouts might cause further state transitions or responses to be produced, or both.

The LLTD protocol allows the LLTD Daemon to supply the mapper with a small amount of information about the system, represented as TLVs. The OSL provides information for these because accessing it tends to be an operating system–specific operation.

Porting to Other Environments

The OSL subsystem is responsible for hiding the specifics of setting up raw Ethernet sockets, discovering information about the host and interface for inclusion as TLV properties, enabling promiscuous mode, becoming a backgrounded daemon, recording the daemon's PID in a *pidfile*, and dropping root privileges.

If you are porting to a Unix-like (POSIX) operating system, you should be able to just write a new *osl-FOO.c* file that implements the API loosely documented in *osl.h*. It is likely that some code can be reused from *osl-linux.c*.

If you are porting to a non-Unix environment then you may also need to change *event.c* and the tracing support for syslog in *util.c*.

The file *event.c* is a wrapper around **select()**, managing a list of functions that should be called at particular times, and socket fds, which should be watched for data becoming available to read. The daemon is single threaded for simplicity, and *event.c* is where asynchrony between I/O and timeouts is handled. If you are merging this code with yours where you already have a main select loop, then you must substantially modify *event.c* to call down to your select loop API.

The *osl-linux.c* port uses POSIX.1e capabilities to drop privileges, rather than changing the UID to (for example) nobody. This is so that the LLTD Daemon can retain CAP_NET_ADMIN, which is required to enable or disable promiscuous mode. Ports to systems without capabilities must assess how best to drop privileges while still retaining the ability to enable and disable promiscuous mode, plus the ability to query interface parameters.

Almost all memory allocations are done in *main.c* when starting up; the only dynamic allocations are by *event.c* as it manipulates the linked list of pending timeout events. Except for a **strdup()** in *main.c*, all allocations go through **xmalloc()** and **xfree()** wrappers in *util.c* if you need to customize or instrument the memory allocation. Nothing larger than 160 bytes is allocated on the stack in one allocation, although obviously the maximum possible stack size will be much larger than this.

Floating point is used only in **random_uniform()** in *util.c*: you might be able to remove this dependency on floating point—that is, for porting to kernel or other minimalist environments—with suitable knowledge of your platform's **rand()** implementation.

Typical TLVs provided by the LLTD Daemon, sorted by importance are:

- *Bssid*
  This is the 6-byte hardware (MAC) address of the wireless access point with which the interface is currently associated.

  Required: You must provide this TLV if appropriate.
- *hostid*
  This is a 6-byte identifier unique to this host; usually the lowest hardware (MAC) address in the system. This allows a mapper to notice that multiple LLTD Daemon instances are actually running on the same host—for example, if there are multiple network interface cards (NICs) in the machine.

  Recommended: You should provide a valid hostid TLV if your platform might have multiple NICs.

Note 1: For definitions of other important TLVs that the device should report, including physical_medium, wireless_mode, and link_speed, refer to the *Link Layer Topology Discovery Protocol Specification*, located at http://www.microsoft.com/whdc/Rally/LLTD-spec.mspx.

Note 2: In this discussion of porting, the following general issues should be considered:

- Endian sensitivity.

  The example code was developed on little-endian, for little-endian. Therefore, in some places structures are used directly in parsing received packets. Usually, packet structure components are treated as units and no endian-sensitive operations are performed on saved copies. By their nature, TLVs are often reconstructed, so care has been taken to ensure that their internal and wire formats are consistent.

- Structure packing assumptions.

  All structures are packed to byte boundaries, and no padding is expected.

Code Organization

Figure 3 illustrates the general flow of control in LLTD Daemon, similar to the data flow among the code modules.

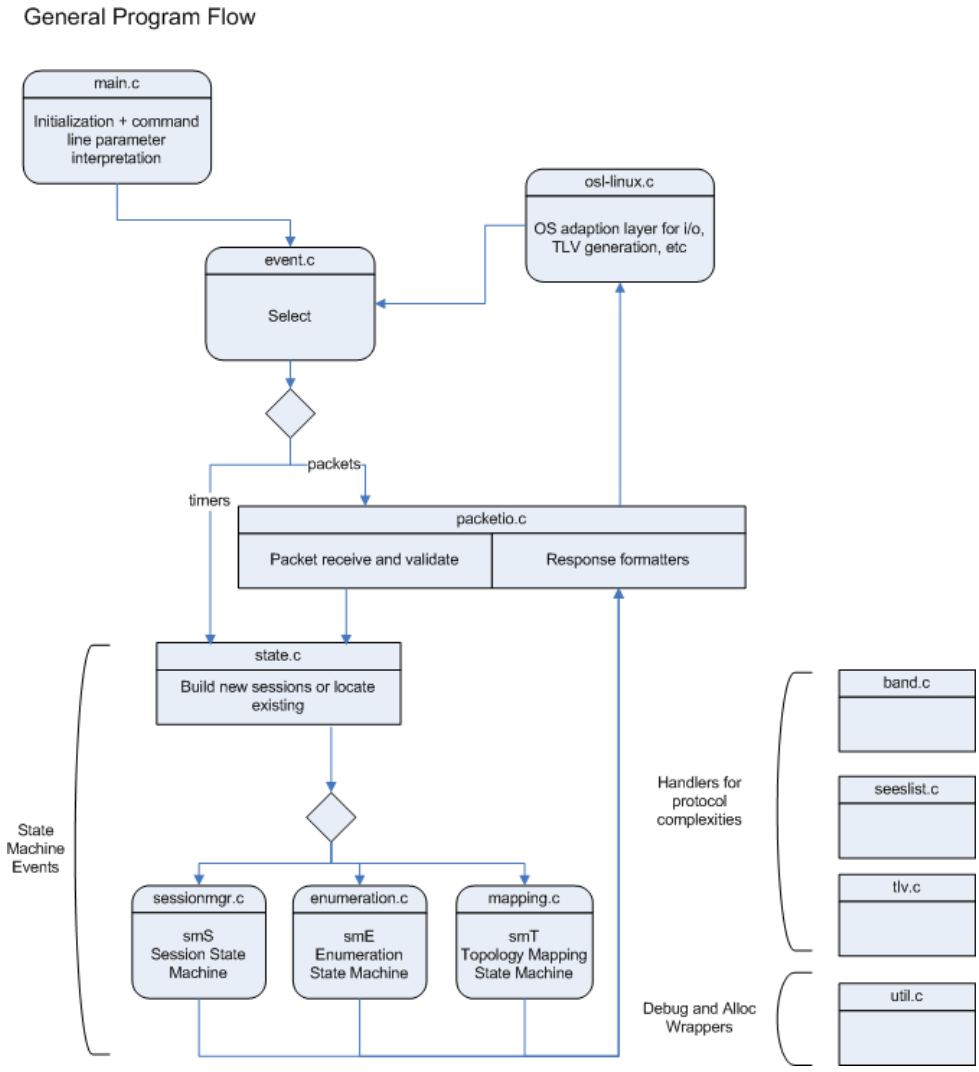


Figure 3: General Control Flow in LLTD Daemon

In overview, the LLTD Daemon is a synchronous network protocol daemon that is running in user space. After initializing and daemonizing, the program waits in a select statement for process-level events, including the receipt of LLTD protocol packets at the raw socket (through a porting wrapper) and various timeouts on both a per-process and per-session basis.

Each of those events undergoes basic validation in the low-level packet I/O routines, and then the events are refined into more distinct state-machine events. These latter events are presented as input for transitions in the three separate state machines and usually trigger the sending of a response packet.

Response packets are formatted as they leave, again in the low-level packet I/O routines, and sent through the operating system–specific porting wrapper to the wire. Returning from the transmit porting wrapper, the code returns to its select, to close the loop and await more work.

Invoking the LLTD Daemon

In normal operation, the LLTD Daemon is invoked with one parameter: the interface name on which it is to listen. This could be eth2, wl0, or br0, which is the case with the WRT54GS (a bridge that combines eth1 and wl0 on the Linksys device).

For debugging purposes, two flags are available and are placed in arbitrary order preceding the interface name:

- The appearance of **-d** instructs the program not to daemonize, which is helpful when you want to see debugging output immediately rather than reading it from a log.
- The appearance of **-t** <*decimal-trace-flags*> qualifies which aspects of the processing will be enabled for tracing, assuming, of course, that the program was compiled with debugging enabled. Invoking the program without arguments prints a summary of the invocation parameters and exits.

  The <*decimal-trace-flags*> are a decimal representation of a bitwise or of the five supported trace-partitioning flags:

  TRC_BAND = 0x01
  TRC_PACKET = 0x02
  TRC_CHARGE = 0x04
  TRC_TLVINFO = 0x08
  TRC_STATE = 0x10

So, for example, tracing packets and state transitions would require the following:

0x02 | 0x10 = 0x12 = 18 (decimal)

The program also has a small configuration file (*lld2d.conf*) that is placed in the /etc directory, to specify the name of the icon file that it should return as TLV 14. The icon file's name is arbitrary, although the file must contain a 48x48 pixel icon to be properly displayed. By default, the icon file path is /etc, but the path can be included in the name.

Two icons are included with the example code:

- *wrt54g.ico,* which represents a generic Router and AP
- *tux.ico* for an x86 PC

Move one of those icons (or your choice of 48x48-pixel full-color Windows icon files) to the location, and then name that as the indicated configuration file. Then watch the unit appear on a map.

Hotspots for Customization

The following are key issues to observe in customization.

- Memory allocation and debug/warning wrappers.

  The file *util.c* contains wrappers for all heap allocation, as well as wrappers for reports of debug, warning, or failure messages.

- I/O details and TLV **get**_<*tlv-name*>**()** routines.

  Most of the operating system–specific customization is done in *osl-XXX.c*. In the example code, this is *osl-linux.c*. This module abstracts the handling of privileges and capabilities, PID-file management, socket configuration for promiscuous capture, socket-IO, and daemonizing to release all consoles.

  The second half of the code in this module is concerned with TLVs. There is a skeleton **get**_*XXX***()** routine for each of the currently defined TLVs. Most of these are just skeletons and thus return:

  TLV_GET_FAILED

  This return message forces the TLV engine to ignore the TLV in both Hello packets and QueryLargeTlvResp. After saving their TLV data through the common pointer parameter, the ones that actually generate a value, whether from complex code or just from a constant, return:

  TLV_GET_SUCCEEDED

  These **get**_*XXX***()** functions are the main point of customization. Choose the TLVs to be returned, modify the matching get_... routine, and check that a corresponding **write**_*YYY***()** routine exists for the datatype. If you do not see the routine, look in the second half of *tlv.c*.

- TLV functionality.

  Usually customization of TLVs is done to **get**_<*TLV-name*>**()** routines in *osl-linux.c*. Another place for customization is the file *tlvdef.h*, which contains the table of TLVs. The manufacturer uses this table—which contains many macros—to indicate inclusion in the Hello packet and decisions to generate once (at startup) or each time the TLV is requested.

  For more information about this table, see "Frequently Asked Questions" later in this paper.

- Iconic representations in the map.

  A manufacturer can provide a full-color icon in Windows .ico format, place it wherever desired in the file system, and point the LLTD Daemon toward it with the configuration file (*/etc/lld2d.conf*). This icon should not exceed 256 K in size.

Frequently Asked Questions

How is the TopologyMapping service different from the QuickDiscovery service?

The LLTD protocol has been expanded to allow more than one simultaneous session, although it is still restricted to only one topology-mapping session at a time. Up to ten other computers can now issue Discover packets and start sessions, although they must be using the QuickDiscovery service instead of TopologyDiscovery. This allows them to at least discover the existence of the various responders on their segment, capture whatever TLVs the Hello packets provide, and—by eavesdropping on the mapping in progress—even capture some of the actual mapping information for themselves, all without becoming a topology mapper themselves.

QuickDiscovery service users can issue only Discover packets; none of the more complex TopologyMapping functions are available to them.

What are TLVs, and how do I use them?
The TLV handling in this sample implementation is particularly intricate. The file *tlvdef.h* is actually included in three places, with a different interpretation in each place. This design makes it easy to add or modify TLV allocations, by adjusting lines in *tlvdef.h* and letting the complex macro usage define all the necessary stubs, routines, and linkages to support it. If you want to understand TLVs further, look up the token-merging operator ## in C-style macros.

In *tlvdef.h*, each TLV is defined as the arguments to a macro named TLVDEF. For example:

```
/*       C-type,       name,    repeat, nmbr, access-type, inHello */
TLVDEF( etheraddr_t,  hostid,         ,    1, Access_unset, TRUE )
```

Examining each field in turn shows the allocation size of the TLV, its name, any array size, whether it is a fixed-length array, the type of the TLV (*nmbr*), an access type, and a flag that will be TRUE if the TLV is to be included in the Hello response packet.

As discussed earlier, any TLV can be accessed by a TopologyMapper as an LTLV. Only those marked as **inHello** are ever candidates for inclusion in the Hello response, as a small TLV, however. A fixed-size array, such as occurs in the TLV #15 machine_name would have a C-type ucs2char_t and a repeat of [16], which indicates an array of 16 unsigned double-byte characters.

TLVs are processed in one of three ways, indicated by the access type:

- **Access_unset.** This type marks a TLV as one that is calculated a single time (when first asked for) and saved for any future Hello responses. The routine **tlv_write_tlv()** in *tlv.c* calculates the TLV value, saves it in a particular place in the global variable g_info, and marks the associated TLV descriptor as Access_static, assuming that the calculation succeeds. It can fail, for many reasons.
- **Access_invalid.** If you must include a TLV for documentation purposes, but do not want it ever to be part of either a Hello response or a QueryLargeTlvResponse, then mark it as Access_invalid.
- **Access_dynamic.** The third processing style (consider *unset* + *static* as one style) is Access_dynamic, and TLVs marked with this access type are recalculated each time they are asked for.

The actual processing of the TLVs is handled by two sets of routines:

- **get**_ routines, which calculate the TLV values.
- **write**_ routines, which write those values into a Hello or QueryLargeTlveResp packet, when asked.

Each named TLV has a get_ routine, and the macro defines them to include the TLV name in the function name, automatically. For the above example, TLV #1, which is named *hostid*, declares the function **get_hostid()**, which must then appear in the file *osl-linux.c*.

In a similar fashion, another reinterpretation of the TLVDEF macro creates the linkage in the TLV descriptor table Tlvs (lines 321-328 in *tlv.c*) to the various write functions, of which one is defined for each distinct C-type that appears in the *tlvdef.h* file. Thus, TLV #1 (hostid) causes a function **write_etheraddr_t()** to be declared as the prototype for the descriptor's linkage, and this routine appears at

lines 113 through 141 of file *tlv.c*. Because only one routine is required for each C-type, the write_ routines are shared as appropriate, whereas the get_ routines are unique (one for each TLV name).

A structure is created that contains space to store all of the TLV data as it is calculated. This appears as the struct {} tlv_info_t in lines 25 through 30 of *tlv.h*. That struct definition is used to declare the global variable g_info, as discussed earlier. The previously mentioned TLV descriptors are the final use of *tlvdef.h* (see lines 321 through 328 of *tlv.c*). Each TLV has a descriptor in that table, which is the array Tlvs[]. This table of TLV descriptors serves to link the get_ and write_ routines to their respective data storage areas in g_info. The table also holds various flags, including the access types.

What powers this protocol?
The key to a successful implementation of the protocol is to understand the three state machines and their representative diagrams, which are described in the "LLTD: Link Layer Topology Discovery Protocol" specification. The following is a brief description of the state machine implementations in this porting kit.

The protocol processor is divided into three separate state machines, which handle the session status (smS), the enumeration process (smE), and the topology mapping process (smT). Incoming packets and other process-level level events such as timeouts are converted into state machine events and passed to the smX machines, each of which resides in its own compilation module (*sessionmgr.c*, *enumeration.c*, and *mapping.c*, respectively).

- The passing of state machine events is controlled by code in the *state.c* module.
- Low-level packet handling and validation, as well as transmitted-packet formatting, are found in file *packetio.c*, whereas process-level events (timeouts and the select-loop for packet reception) appear in *event.c*.
- The module *main.c* handles the startup, and the module *band.c* handles the multi-BAND responder protocol. TLVs, of course, are the subject of *tlv.c*.
- Because an effort was made to make this code portable, operating system–specific code is confined to *osl-linux.c* and utility help is provided by *seeslist.c* and *util.c*.

The state machines comprise 11 states and nearly 70 transition arcs. That does not include the diagnostics messages.

Is the protocol running in user mode, with no required kernel modules?
That is correct—this is a user-mode daemon.

How do I add a new TLV or change an old one to match my device?
Add a line to *tlvdef.h* for the new TLV, create the **get**_<*newname*>**()** function in *osl-linux.c*, and ensure that there is an appropriate write_ function in *tlv.c*. If this function does not exist (because you used a heretofore unseen C-type for this new TLV), create the **write**_<*new-C-type*>**()** function in *tlv.c*.

Why does the responder retain a list of the Probe packets that it has seen, instead of reflecting them immediately to the mapper?
There are two reasons:

- Scalability. Often Probe packets are flooded over a portion (or all) of the network. If every responder to see a Probe packet reflected it to the mapper, then on a large network there would be a huge implosion at the mapper and very high network load.
- Reliability. The current protocol is designed so that the reliable communication between mapper and responder is simple. If responders sent reflections, then significant complexity would be associated with whether the Probe packet got lost between the sender and the responder or the reflection between the responder and the mapper.

What should I do with this list of MAC addresses?
Your device should send the packets back in order, with the gaps between them as specified.

How long must I store the list?
Your device stores the list until you have finished sending the packets—and sent the acknowledgment, if the emit was sequenced.

What must the responder do when the pause value expires? Can I track only the largest value and then do something?
Your device processes them in order. You need only a timer for the pause value until the next packet in order, which can be zero. The packets must be sent in the order in which they are listed in the emit packet.

Aren't 64 entries of MAC address a lot of data for an embedded device to retain?
You should retain the emit packet only until you have processed it. The largest valid emit packet is limited by the maximum packet charge of 64 packets.

How does the mapper request me to return the list I am retaining?
The list you retain is of Probe packets that you have seen. The mapper sends you a query packet, and you send it a query response packet.

The LLTD Protocol Specification indicates that the responder must emit Train packets to all of the devices on the list. Is this correct? If not, why do I get the pause time?
Your responder sends either Train or Probe packets (as specified in the type field) to the specified addresses. The packets may need to be sent with gaps between them, which is why a pause time is also specified between each packet to be emitted.

Should I track those stations that are sent to me in an Emit packet and then watch for Probe packets that respond to a Query packet with my list?
When a responder is in command mode, it should record in a list all of the Probe packets that it sees, including loopbacks (copies of packets that the responder may itself be sending because of one or more emit commands). It retains all of these until it is later queried by the mapper. These records of Probe packets that were seen are called Recveedescs (receive descriptors).

Note that a responder primarily sends and receives Probe packets. An Emit packet is used to tell it to send, and Query/QueryResp is used to ask it what it has seen.

What is the purpose of "charge" and "flat"?
The purpose of the charge is to convince the responder that the mapper could have done by itself as much "damage" to the network as the responder's transmissions might do by proxy and hence that it is reasonable to send the packets that the mapper has asked it to send. The purpose of the flat is to deal with a situation in which one or more of the charge packets is dropped in the network.

The mapper is not expected to send different amounts of charge for different responders, and a flat is not intended to enable that action.

Terminology

Acronyms

BAND
block adjust node discovery, a fast and scalable node enumeration algorithm.

LLD2
Link Layer Discovery and Diagnostics protocol: an outdated term replaced by "LLTD QoS Extensions."

LLD2D
A synchronous network protocol daemon that is running in user space

LLTD
Link Layer Topology Discovery protocol.

MAC address
Media Access Control address. Each network adapter has a unique address that is typically written as a string of 12 hexadecimal values.

RepeatBAND
An extension to BAND that supports multiple enumerators.

OUI
Organizationally unique identifier, or the three most significant octets of a MAC address as maintained by the IEEE Registration Authority.

PnP-X
An extension of Plug and Play in Windows Vista that allows virtually-connected devices to be integrated into the Microsoft Windows Plug and Play subsystem.

TLV
Type-length value. A property of an interface, so named because each property is composed of a Type field, a Length field, and a value.

UUID
universally unique identifier. A 128-bit value that is assigned to any object and is guaranteed to be unique.

XID
A transaction ID, a 16-bit value. With stable storage, XID values are sequential; without stable storage, XID values are assigned at random.

General

controller
The (arbitrary) station that initiates a network test request.

enumerator
The (arbitrary) station that participates in the node discovery process.

hub
A data link-layer network device that acts as a shared bus. All stations that are connected to a hub are on the same segment, and therefore each station that is connected to a hub sees all frames that are sent to or from all other stations on that hub. Compare with “router” and “switch.”

mapper
The (arbitrary) station that initiates a topology discovery request.

quick discovery
The process of discovering responders on a network.

responder
A client station to which mappers and enumerators send commands by using the LLTD protocol that this document describes.

router
A network-layer device that defines the limit of an Ethernet broadcast domain. Compare with “hub” and “switch.”

segment
A set of stations that all see each others’ frames.

session
A context for managing communication over a protocol to another station identified by a specific MAC address and service pair.

sink
The responder station that is the target of a network test session.

station
An end-system that is connected to a switch, hub, or router.

switch
A data link-layer device that propagates broadcast frames between network segments and allows unicast communication between pairs of stations on different segments. Stations that are connected through a switch see only the frames that are destined for their segment. Flooding (that is, seeing a frame for someone outside the segment) occurs only if the switch has not learned that MAC address yet. Compare with “hub” and “router.”

topology discovery
The process of discovering the topology of a network. Compare with "quick discovery."

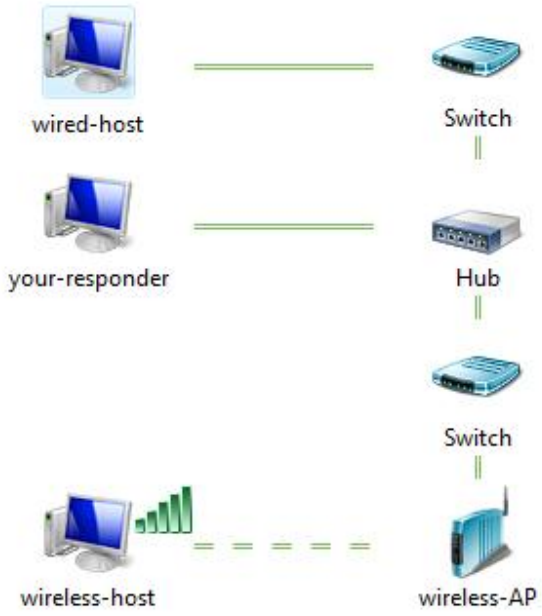
Appendix. Component Table FAQ

Why does the Windows Vista Network Map show more devices than I have?
LLTD can map network topologies and, in doing so, discover all of the logical components in your device. You can collapse all these logical components into one device in the Network Map by using the Component Table TLV. It is strongly recommended that you implement the Component Table TLV so that the internal components of your device are "collapsed," thereby improving the correlation between the networking devices that appear in the Windows Vista Network Map and the actual, physical units that the consumer sees.

How do I construct a Component Table TLV (0x1A)?
First, you must know what logical components are present in your package. This is usually done by connecting at least one Windows Vista PC (or Windows XP PC if you have the optional responder package installed) to the wired *and* wireless links on your device. Note the following:

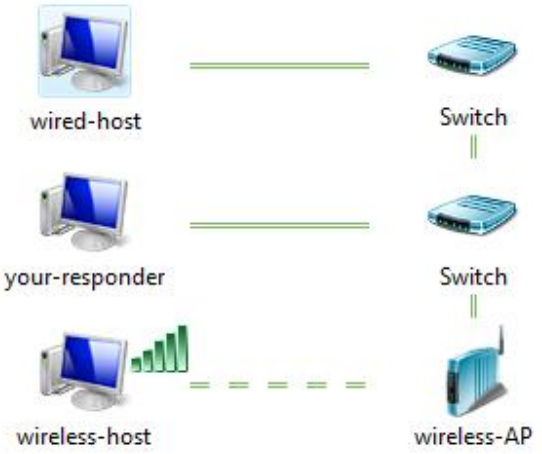
- If your device has a built-in switch, connect to any one LAN port of your choice. LLTD is a nonrouted protocol; information that is obtained about client PCs or devices on the LAN-side of the router or gateway should never be exposed on the wide area network (WAN) interface.
- Ensuring that LLTD responders (Windows Vista PCs) are connected to all of your device's non-WAN interfaces is critical to fully discover the components within your device and thereby create an accurate Component Table.
- Make sure that the interfaces of your Windows Vista PCs are set to "private," not "public" or "domain." By default, Windows Vista suppresses the LLTD responder and topology mapping when an interface is not set to "private" mode. Ensure the correct setting on the **Start** menu by clicking **Control Panel**, and then clicking **Network and Sharing Center**; under the network graphic, select **Customize** and ensure that the connection type is set to **Private**.

Consider, as an example, a typical wireless router device. You should connect a Windows Vista PC to the built-in switch and associate a Windows Vista laptop to the built-in Wi-Fi access point. Next, bring up the Network Map on any of the Windows Vista machines. The map should look like the following examples.



Case 1: Pass-through bridge

In Case 1, your device has a bridge that behaves like a hub; the responder sees all packets that pass through it. The switch located between the bridge and the wireless AP must be ignored because LLTD always assumes that it is impossible to connect a hub directly to a wireless AP without going through a switch. This assumption would have been true if LLTD were mapping a network with such a physical layout.



Case 2: Switching bridge

In Case 2, your device has a bridge that behaves like a switch; the responder sees only packets that are destined for it.

Knowing what type of bridge you have in your device is the most important part of constructing the component table. If your bridge is a hub, the binary payload of your component table would look similar to the following:

01 00 00 01 00 01 0A 00 6C 02 01 AA BB CC DD EE FF 02 04 00 0F 42 40

To dissect:

01 00	The header.
00 01 00	The bridge component with hub behavior. (Note: The "internal hub-switch" behavior as documented in the specification is used only to identify special cases such as the Windows Network Bridge introduced in Windows XP and some wireless bridges in the market.)
01 0A 00 6C 02 01 AA BB CC DD EE FF	Wireless radio, 54 Mbps, 802.1g, infrastructure mode, BSSID = aa:bb:cc:dd:ee:ff.
02 04 00 0F 42 40	A built-in switch operating at 100 Mbps.