**IEEE P802.15**

**Wireless Personal Area Networks**

|  |  |
| --- | --- |
| Project | IEEE P802.15 Working Group for Wireless Personal Area Networks (WPANs) |
| Title | **ULI 6TiSCH “Hello World” Example** |
| Date Submitted | [9 July 2018] |
| Source | [Randy Turner][<Landis+Gyr>][] | Voice: [ ]Fax: [ ]E-mail: [randy.turner@landisgyr.com] |
| Re: | TG12 Example ULI Application Bootstrap |
| Abstract | [Description of ULI “stack” bootstrap with corresponding “Hello World” Application] |
| Purpose | [Vetting initial 802.15.12 proposals against potential real-world implementation] |
| Notice | This document has been prepared to assist the IEEE P802.15. It is offered as a basis for discussion and is not binding on the contributing individual(s) or organization(s). The material in this document is subject to change in form and content after further study. The contributor(s) reserve(s) the right to add, amend or withdraw material contained herein. |
| Release | The contributor acknowledges and accepts that this contribution becomes the property of IEEE and may be made publicly available by P802.15. |

**6TiSCH Minimal “Hello World” Application**

This document provides an example of how to use the currently proposed 802.15.12/ULI subsystem to send and receive an IPv6/IETF 6TiSCH "Hello World" packet. The text specifically shows how an implementation might work from system boot through applications sending and receiving a “hello world” packet. The example(s) included in this document are loosely based on a Unix/Linux network/system programming model.

Using the current ULI “overview” illustration, this example would use the following components:



This example document defines a set of ULI profiles to be used for the IETF 6TiSCH "Minimal" operational mode. In the subsequent text of this document, anywhere the term "profile" is used, it is assumed to be the following profile definition(s).

 802.15.4 MAC

 - FrameVersion=2

 - Node Extended Address (EUI-64)

 - TSCH

 - Slot Width (10 msec)

 - Timeslot Template

 - Enhanced Beacon Period

 - Max Retransmissions

 - Default 802.15.4 TSCH hopping sequence

 - macTimeslotTemplateId=0

 - MAX\_EB\_DELAY

 - MAX\_NEIGHBORS\_TO\_WAIT

 - MAX\_TSN\_SILENT

 802.15.4 PHY

 - 2.4 Ghz

 - O-QPSK

 802.15.4 Security

 - Key(s)

 K1 (EB Authentication)

 K2 (DATA and ACK Authentication)

 (In this example, keys are pre-configured, NOT learned)

 - Other relevant PIB values

 - Authentication required, encryption optional

 6Tisch Module

 - Number of scheduled cells

 - Number of unscheduled cells

 - Cell options

 - Network Name

*At a previous plenary meeting, we talked about the requirement for (potentially) multiple profiles to be configured (protocol-module-specific, MAC/PHY, security, etc.). Therefore, the PDE-MGMTCREATE function will probably be called multiple times, and if the PDE-MGMTCREATE function can target different subsystems with a profile, there may need to be an additional parameter that identifies which subsystem should be configured with the specified profile.*

*I looked at the PDE-MGMT-COMBINE request to accomplish the "multiple profile" requirement, however, I think this function needs further explanation, with example(s).*

**Boot Sequence (example, not normative)**

This sequence is loosely based on a Unix/Linux network device bootstrap that employs a “bottom up” initialization method.

1. The 802.15.12 driver is initialized, causing the built-in mandatory management module to also be initialized.
2. The 802.15.12 driver will enumerate any non-mandatory protocol modules from a statically defined module list. The 802.15.12 driver will call the "init" entry point in each protocol driver in the static list.

*The 802.15.12 driver also contains a "protocol module register/deregister" API to allow protocol modules to be dynamically added and removed from the system without rebooting/rebuilding the "stack". In this example, the only static module definition is a descriptor for the “6TiSCH” protocol module.*

1. 802.15.4 Device (Transceiver) Discovery [ One transceiver found = /dev/rf/0 ]
2. For each RF device found, a separate PDE “context” is created, identified by a

specific device index (/dev/pde0, pde1, pde2, etc.). In this case, a single radio

causes “/dev/pde0” to be created.

1. The 802.15.12 driver will also create an MMI context in which the pde0 driver instance will operate – in this example, this MMI context is “mmi0”

 *At this point, we have a stack that looks like this:*

 */dev/pde0 (PDE API)*

 *(6TiSCH protocol module)*

 *mmi0 (MMI API)*

 */dev/rf/0 (802.15.4 MAC/PHY API)*

1. In addition to the creation of the /dev/pde0 pseudo device being created, the 802.15.12 driver will also create a corresponding “/dev/pde0m” pseudo device. “/dev/pde0” is the PDE-DATA interface, and pde0m is the PDE-MGMT interface. They are separate pseudo devices in this example because the management interface device may (probably does) have different requirements for security / access controls.

*The stack outline above is created for each RF transceiver found in the system hardware configuration. There is one copy of each of the modules/driver code, which is multiply instanced per RF transceiver. /dev/pde1 might represent another stack context that is associated with another radio device (/dev/rf/1 for example).*

*For the purposes of this example document, the 6TiSCH protocol module also contains 6LoWPAN functionality, as well as 6top functionality. This is a bit different than the 802.15.12 "Overview" illustration showing separate modules for 6LoWPAN and 6top.*

1. When the 6TiSCH protocol module "init" entry point is called during 802.15.12 driver startup, the 6TiSCH module will call the "register" API within the 802.15.12 driver, essentially notifying the 15.12 driver of the modules' interest in ethertype A0ED. The 802.15.12 driver will use this ethertype to vector traffic to/from the 6TiSCH module and the PDE/MMI subsystems.

For this ULI "Hello World" example, the 2 nodes involved in the "Hello World" exercise will use link-local addressing.

Receiver: fe80::8c45:ff:fe26:3c1

 Sender: fe80::8c45:f1:fd25:3d1

Both sender and receiver applications perform the following management sequence. The purpose of this management sequence is to create and activate the profiles necessary for 6TiSCH operation, and to establish security keys and initiate the “network join” procedure for 6TiSCH. This sequence of management commands might be executed from some type of CoMI or other management application that wants to configure the ULI and/or MAC/PHY based on a description of manageable objects (YANG, MIB, etc.):

 **MGMT Sequence**

 int fd;

 char \*netname = "mynetwork";

 struct {

 ccm\_key\_t k1;

 ccm\_key\_t k2;

 } keylist;

 fd = open("/dev/pde0m");

ioctl(fd, IOC\_PDE\_MGMTCREATE, 6t-macphy-profile, sizeof(6t-macphy-profile));

ioctl(fd, IOC\_PDE\_MGMTRCVEXEC, 6t-profile-id, sizeof(6t-profile-id));

// “setkey” for k1 and k2, according to 6T spec

 ioctl(fd, IOC\_6T\_SETKEY, keylist, sizeof(keylist))

 ioctl(fd, IOC\_6T\_JOIN, netname, strlen(netname));

If the PDE receives a management request (IOCTL command) with a function code it does not understand, it will call the protocol modules' management interface with the management request.

When the 6TiSCH protocol module receives and activates a profile (MGMTRCVEXEC), it will create a network interface (for example, “rf0”). Network interfaces are different than standard “device” interfaces in that they provide I/O entry points and configuration data required by the IP stack. This interface is subsequently visible to standard network configuration and management tools in the operating system. The following line from the network “bringup” script makes the network interface available to applications.

 iface rf0 inet static

The initialization of the network interface (rf0) will open a file descriptor to /dev/pde0 and this file descriptor will be stored in the rf0 network interface descriptor for later I/O operations.

rf0->devfd = open(“/dev/pde0”)

Also, the IPv6 stack needs to know the MTU for this new interface. It can accomplish this by querying the underlying PDE device.

Ioctl(rf0->devfd, IOC\_6T\_GETMTU, &rf0->mtu, sizeof(rf0->mtu))

The network interface “rf0” will remain in an OFFLINE state until the network join process has completed. The network join process for 6TiSCH briefly involves the following:

* Listening for enhanced beacons (EBs) advertising the network identified in the IOC\_6T\_JOIN message
* Authenticating any received EBs
* Running an algorithm that utilizes the “join priority” specified in the EB to decide on a “join assistant” and/or time-source neighbor.

Once the node has fully joined the network, the 6TiSCH protocol module might provide the beaconing service (for other joining nodes on the network) through the services of the MMI, as per the following example:

6TiSCH EB:

 MMI-DATA.request(SrcAddrMode = extended,

 DstAddrMode = short

 DstPanId,

 DstAddr = (0xffff)

 Msdu,

 MsduHandle,

 HeaderIeList = None,

 PayloadIeList =

TimeSyncIE,

TimeSlotIE,

ChannelHoppingIE,

TSCH Slotframe and Link IE

 AckTx = False,

 SecurityLevel,

 KeyIdMode, (Security parameters driven by security profile)

 KeySource,

 KeyIndex,

 DataRate,

 PanIdSuppressed = True,

 SeqNumSuppressed = False,

 )

At this stage, our “stack” pictures looks like:

 “rf0” (OS network interface)

 /dev/pde0 (PDE API)

 (6TiSCH protocol module)

 mmi0 (MMI API)

 /dev/rf/0 (802.15.4 MAC/PHY API)

**Send Example**

 struct sockaddr\_in6 dst;

 char \*pktout = "Hello World!";

 dst.sin6\_family = AF\_INET6;

 dst.sin6\_port = 11000;

 dst.sin6\_flowinfo = 0;

 dst.sin6\_scope\_id = 3;

inet\_pton("fe80::8c45:ff:fe26:3c1", (void \*)&dst.sin6\_addr, sizeof(dst.sin6\_addr));

 s = socket(AF\_INET6, SOCK\_DGRAM, IPPROTO\_UDP);

 sendto(s, pktout, strlen(pktout), 0, &dst, sizeof(dst));

 .

 . down through socket layer to IPv6

 .

/\* IPv6 will form the full packet (ipv6 hdrs + udp hdrs + app data) and send the fully formed IPv6 packet to the rf0 interface. If IPv6 has determined that the outbound packet is larger than the MTU of the RF interface (rf0->mtu) then the following “write” operation will occur for as many fragments as needed. \*/

 Ip\_out(rf0, pkt, sizeof(pkt))

/\* The rf0 interface will translate this operation into a PDE operation based on information captured when the rf0 interface was created \*/

 . dstAddr = Neighbor discovery for link-local address

 . dstProtocolID = rf0->protID (A0ED)

 . pdeData = IPv6 + UDP headers + application message ("Hello World")

 . pdeHandle = an opaque transaction identifier

 .

 PDE-DATA.request(dstAddr, dstProtocolID, profileID, pdeData, pdeHandle)

*I am proposing that only 1 MAC/PHY profile set can be active at any one time. So the fact that we have previously activated our profile means specifying (again) the profileID in the PDE-DATA request is redundant, or at a minimum, we could make a PDE DATA.request with a "NULL" profileID, which would mean "use the active MAC/PHY profile"*

Based on the dstProtocolID (A0ED), the PDE routes the outbound packet to the 6TiSCH protocol module. The 6TiSCH protocol module will perform 6LoWPAN IP header compression on the outbound IP packet, as well as potentially fragmenting the outbound packet, prior to delivering packets to the MMI subsystem.

The PDE subsystem extracts all relevant information from the active MAC/PHY profile(s) and creates an MMI-DATA.request:

 MMI-DATA.request(SrcAddrMode = extended,

 DstAddrMode = extended(long)

 DstPanId = (PAN ID from join assistant)

 DstAddr = (dstAddr from PDE request)

 Msdu = Packet,

 MsduHandle = opaque transaction ID,

 HeaderIeList = None,

 PayloadIeList = None,

 AckTx = False,

 SecurityLevel,

 KeyIdMode, (Security parameters driven by security profile)

 KeySource,

 KeyIndex,

 DataRate = 1Mbps,

 PanIdSuppressed = True,

 SeqNumSuppressed = False,

 )

*One thing to note about the MMI.data request – The “AckTx” parameter is set to “False”, basically indicating to the receiver that a MAC-layer acknowledgment is not required. 6TiSCH is IPv6-based, and TCP/IP and UDP/IP (as well as other IP transports) are designed to fill in the reliability requirement. TCP/IP and UDP/IP are designed to work whether the MAC layer is reliable or not. Of course, if you’re running something other than IP, then an ACK might be convenient to indicate reception by the receiver. Running without ACKs may provide some bandwidth savings and/or free up some of the tight realtime requirements necessary for intra-slot timing in TSCH mode. Of course, some routing protocols (i.e., RPL) might use “ACK not received” statistics for optimizing its’ routing algorithms – however, there are scenarios with a 6TiSCH Minimal deployment that do not necessarily need active routing protocols. When running a routing protocol that needs inter-node transmission performance statistics, the “ackTx” parameter would be set to “True”.*

*When MAC layer ACKs are used with the 6TiSCH protocol module, the 6TiSCH protocol module would forward any “CONFIRM” or “ERROR” messages regarding transmission performance to some external component that can use these statistics for optimizing routing or other function.*

 MCPS-DATA.request (same as MMI parameter list)

**Receive Example**

 MMI-DATA.indication(...)

 MMI routes the inbound packet to the 6TiSCH protocol module based on

 the received ethertype (A0ED)

6TiSCH protocol module performs 6LoWPAN decompression and/or defragmentation, and delivers the packet to PDE.

 PDE-DATA.indication(srcAddr, dstAddr, dstSapId)

 srcAddr = Extended MAC address of sending node (above)

 dstAddr = extended MAC of receiver

 dstSapId A0ED (delivered by 6TiSCH module to IPv6)

The IPv6 “indication” callback demultiplexes the packet and delivers the packet to the UDP module. The UDP module demuxes based on UDP port and delivers the inbound packet through the socket layer to the receive application.

int s;

 struct sockaddr\_in6 src;

 struct sockaddr\_in6 bindaddr;

 socklen\_t socklen = sizeof(src);

 bindaddr.sin6\_port = 11000;

 bindaddr.sin6\_family = AF\_INET6;

 bindaddr.sin6\_addr.s\_addr = IN6ADDR\_ANY;

 bindaddr.sin6\_scope\_id = 3;

 s = socket(AF\_INET6, SOCK\_DGRAM, IPPROTO\_UDP);

 bind(s, &bindaddr, sizeof(bindaddr));

sts = recvfrom(s, &rcvmsg[0], sizeof(rcvmsg)-1, 0, (struct sockaddr \*)&src, &socklen);

 if (sts > 0) {

 rcvmsg[sts] = '\0';

 printf("%s\n", rcvmsg);

 }