# BLIP Tutorial

From TinyOS Wiki

Error creating thumbnail: convert: no decode delegate for this image format `/tmp/magick-zR7f0RNk' @ error/constitute.c/ReadImag convert: missing an image filename `/tmp/transform_662884483 1.png' @ error/convert.c/ConvertImag

## Contents



# BLIP Tutorial

BLIP, the Berkeley Low-power IP stack, is an implementation in tinyos of a number of IP-based protocols. Using blip/tinyos, you will be able to form multi-hop IP networks consisting of different motes communicating over shared protocols. Due to the rapid evolution of the relevent IETF and IEEE standards, blip is currently not completely standards compliant; however, it does provide significant interoperability with other IP networks.

## Goal and Prerequisites

In this tutorial, you will learn how to install and configure a blip-based IPv6 subnet. You should already have a working tinyos environment installed for any platforms you plan on using. Additionally, a Linux device or virtual machine image with standard tools (gcc, autoconf) is necessary for the edge router.

- **The blip router is only supported on Linux, as of 2.1.1.**
- **The latest version of TinyOS has a new implementations of many protocols. See BLIP 2.0**

## 6lowpan/IPv6 Basics

A complete introduction to IPv6 network topology and addressing are beyond the scope of this tutorial; unfamiliar readers may wish to familiarize themselves with some of the concepts at play here.

hosts *vs.* routers: in an IP network, devices may participate as *hosts* or *routers*. Generally, *hosts* do not forward packets or participate in routing protocols, while *routers* do. Every mote running the blip network stack functions as a router and is capable of forwarding packets and making routing decisions. Motes thus form a multihop IP subnetwork.

subneting: The smallest unit of network management in IPv6 is the subnet. In blip, a subnet consists of a number of motes (*node routers*) and one or more higher function devices (*edge routers*) which perform a number of other routing functions for the network. The edge routers also may provide routing to other networks. In terms of network topology, these blip subnets are generally configured as stub networks.

MTU: the Maximum Transfer Unit refers to the largest payload which may be sent in a link-layer data frame. Most blip node routers use the IEEE 802.15.4 physical layer, which limits the MTU to around 100 octets. To provide the larger payloads to upper-layer protocols, blip implements 6lowpan "layer 2.5" fragmentation; as a result, the maximum size IP-layer datagram is 1280 octets.

# Configuration Overview

Commissioning a simple blip subnetwork is not complicated. Here are the steps:

- install and activate an edge router
- (optional) configure routing to other networks
- begin activating blip node routers

## Platform Support

Blip provides an IPv6 layer on top of `6lowpan` and IEEE 802.15.4. Since it does not use the TinyOS Active Message layer, a small amount of radio support is needed to provide a separate message dispatch mechanism, and arbitrate between the AM and Ieee154 layers; this allows applications to include both IP and AM-based protocols.

| target | radio | status |
|---|---|---|
| epic | cc2420 | primary target |
| telosb | cc2420 | supported |
| iris | rf230 | supported |
| micaz | cc2420 | supported<ref>with less buffering</ref> |
| shimmer | cc2420 | compiles, status unknown |

<references/>

## Step 1: Install an Edge Router

Note: the edge router implementation is currently supported for all Linux-based platforms; it is also possible to cross-compile for use on embedded devices. Windows/cygwin support is forthcoming.

This step should be performed on the device to be used as the edge router; typically a laptop or embedded Linux device. In the later case, it is possible to cross-compile the driver; see the "Additional Reading" section for more information"

Steps:

- Add an 802.15.4 interface to the device. This is typically accomplished using a mote programmed with a radio-serial bridge.
- Compile and run the routing driver.

The application used to provide an IEEE 802.15.4 interface is located in `$TOSROOT/apps/IPBaseStation`. Go to that directory, and install the image on an attached mote.

```
$ cd $TOSROOT/apps/IPBaseStation
$ make <platform> blip install
```

Depending on your platform, you may need some extra arguments. For instance, to install to an attached micaz, I do

```
$ make micaz blip install mib510,/dev/ttyUSB0
```

If you are using the Flying Camp Design (http://www.flyingcampdesign.com/) programmer, you also need the `miniprog` extra.

Next, we need to build the routing driver. It has one dependency, the tinyos serial library (libmote.a). To build it, do the following.

```
$ cd $TOSROOT/support/sdk/c/sf
$ ./bootstrap
$ ./configure
$ make
```

If bootstrap fails, you may need to install the necessary automake/autoconf packages for your distribution.

Next, repeat essentially the same steps for the driver, in the $TOSROOT/support/sdk/c/blip directory.

```
$ cd $TOSROOT/support/sdk/c/blip
$ ./bootstrap.sh
$ ./configure
$ make
```

If these steps succeeded, you have successfully built the routing driver for the edge router; the binary is in `driver/ip-driver`.

You can begin running the driver using the following:

```
$ sudo driver/ip-driver /dev/ttyUSB1 micaz
```

If you installed the IPBaseStation image on a different platform, use the appropriate platform or baud rate here.

**Edge Router Configuration**

The routing driver loads its configuration from a file when it starts. The default file is located in `$TOSROOT/support/sdk/c/blip/serial_tun.conf`. This file controls the IPv6 prefix used for the subnet, the address of the router on that subnet (both are taken from the '`addr`' directive), as well as the 802.15.4 channel used. The channel set here must be the same as the channel used by node routers. The default file is shown below; it's a good start for experimentation.

```
# Before you can run the adaptation layer and router, you must
# configure the address your router will advertise to the subnet, and
# which network interface to proxy neighbor advertisements on.
#

# set the debug level of the output
# choices are DEBUG, INFO, WARN, ERROR, and FATAL
# log DEBUG

# set the address of the router's 802.15.4 interface.  The interface
# ID must be a 16-bit short identifier.
addr fec0::64

# the router can proxy neighbor IPv6 neighbor discovery on another
# interface so that other machines on the subnet can discover hosts
# routing through this router.  This specifies which interface to proxy
# the NDP on.
proxy lo

# which 802.15.4 channel to operate on.  valid choices are 11-26.
channel 15
```

## Step 2: Install and Activate Node Routers

Once an edge router has been commissioned, you can begin installing node routers. A sample application is provided in `$TOSROOT/apps/UDPEcho`. This application provides a UDP echo service on port 7, as well as a very simple UDP-based shell on port 2000. To install this application on an attached mote,

```
$ cd $TOSROOT/apps/UDPEcho
$ make <platform> blip install.ID
```

Again, you may need to add additional make flags depending on your target. Also, ensure the channel specified in the Makefile is the same as the channel being used by the edge routing driver.

Once installed, the newly installed mote should check in soon with the edge router. Its IPv6 address is formed by taking the prefix specified in serial_tun.conf and appending the node ID specified when the mote image was installed. For instance, if the prefix specified in the config file was fec0:: and the node id as 1, the mote would have address `fec0::1`.

You may need to install additional packages on your system to get utilities like `ping6`, `tracert6`, and `nc6`. Once you have done so, you should be able to verify the link to your newly installed node:

```
$ ping6 fec0::1
$ tracert fec0::1
$ nc6 -u fec0::1 2000
```

The final "nc6" command is merely a command pipe to the node; type "help" to get a list of commands provided by that mote.

# Additional Topics

This basic tutorial to this point has covered installing an instance of a blip network. The next step is to begin developing your own applications.

## Routing Driver Shell

The edge-routing driver provides an interactive shell on the console and over TCP port 6106. You may connect to the latter using telnet.

```
$ telnet localhost 6106
Trying 192.168.1.100...
Connected to 192.168.1.100.
Escape character is '^]'.
Welcome to the blip console!
 type 'help' to print the command options
blip:openmesh000> stats
Up since Sat Jan  1 00:00:26 2000
  receive  packets: 315162 fragments: 315361 bytes: 29719776
  transmit packets: 93628 fragments: 182000 bytes: 13839131
  forward  packets: 41
blip:openmesh000> conf
configuration:
  router address: 2001:470:8172:21::64
  proxy dev: br-lan
  channel: 21
  version: $Id: config.c 1349 2009-02-16 23:23:19Z stevedh $
blip:openmesh000>
```

Type "help" for a list of commands. Some of the most useful ones are listed here.

- **links**: the driver maintains the link state of the network. This displays the link state reported by each node router.
- **rebuild**: topology information occasionally becomes very stale and out of date. This command rebuild routing state across all attached node routers from scratch. Use sparingly.
- **routes**: packets from the edge router destined to a node router are source routed. This command displays the cached routes that will be used.
- **stats**: show counts of the number of packets forwarded.
- **conf**: print the routers configuration

## Low-power Listening

You might want to run blip over a different MAC layer for better performance or lower power consumption. The `cc2420` stack in TinyOS includes "low-power listening" as an alternative to the default CSMA MAC. To enable it in blip, there are two steps:

- enable LPL on the border router
- enable LPL on all node routers

First, add the following snippet to all application makefiles (you may want to `include` it). This includes the `IPBaseStation` image; you may want to change the sleep interval here to set a different network duty cycle.

```
# the sleep interval needs to be set the same for all participating devices
CFLAGS += -DLOW_POWER_LISTENING -DLPL_SLEEP_INTERVAL=512

# also modifiy blip's L2 parameters to reduce media overload
CFLAGS +=-DBLIP_L2_RETRIES=2 -DBLIP_L2_DELAY=2048
```

Second, recompile the routing driver to enable lpl, as shown below. This changes the number of retransmissions; fewer should be used with using LPL.

```
$ cd $TOSROOT/support/sdk/c/blip
$ make distclean
$ ./configure --enable-lpl
$ make
```

When using LPL, blip will show more variation in round trip times due to the way the protocol works. It may also exhibit poor performance in dense deployments or with high data rates; in fact, even relatively modest data rates can easily overwhelm the medium if you are running at low duty cycles. It is usually important to conduct a "back of the envelope" capacity planning

exercise to see if your deployment will be successful at the predicted data rates, densities, and duty cycles.

# Programming Interface

blip is designed to allow you to easily build your own IP-based applications. This section of the tutorial covers some of the important basics.

## Addressing Structures

The ip-stack provides a bare IP datagram interface to the network layer; this is documented in comments in the code.

For the purposes of socket programming, two data structures are most important. The 'struct sockaddr_in6' and the 'struct in6_addr'. They are substantially shared with the linux/bsd versions, and reproduced below.

```
#include <ip.h>

struct in6_addr
  {
    union
      {
      uint8_t   u6_addr8[16];
      uint16_t u6_addr16[8];
      uint32_t u6_addr32[4];
      } in6_u;
#define s6_addr                 in6_u.u6_addr8
#define s6_addr16               in6_u.u6_addr16
#define s6_addr32               in6_u.u6_addr32
  };

struct sockaddr_in6 {
  uint16_t sin6_port;
  struct in6_addr sin6_addr;
};

/* parse a string representation of an address */
void inet_pton6(char *addr, struct in6_addr *dest);

/* stringify a packed ipv6 address */
int  inet_ntop6(struct in6_addr *addr, char *buf, int cnt);
```

### Usage

Example 1: Suppose we want to setup a sockaddr_in6 to point to ff02::5, port 10000:

```
{
  struct sockaddr_in6 sa6;
  inet_pton6("ff02::5", &sa6.sin6_addr);
  sa6.sin6_port = htons(10000);
}
```

Example 2: Do the same thing, but without the overhead of storing and parsing the string address representation.

```
{
  struct sockaddr_in6 sa6;
  memset(&sa6, 0, sizeof(struct sockaddr_in6));
  sa6.sin6_addr.s6_addr16[0] = htons(0xff02);
  sa6.sin6_addr.s6_addr[15] = 5;
  sa6.sin6_port = htons(10000);
}
```

## UDP

blip provides a UDP sockets layer as a basic application transport service. The UDP interface is located in

```
tos/lib/net/blip/interfaces/UDP.nc
```

and is simple:

```
interface UDP {

  /*
   * bind a local address.  to cut down memory requirements and handle the
```

```
   * common case well, you can only bind a port; all local interfaces are
   * implicitly bound.  the port should be passed in host byte-order (is
   * this confusing?
   */
  command error_t bind(uint16_t port);

  /*
   * send a payload to the socket address indicated
   * once the call returns, the stack has no claim on the buffer pointed to
   */
  command error_t sendto(struct sockaddr_in6 *dest, void *payload,
                         uint16_t len);

  /*
   * indicate that the stack has finished writing data into the
   * receive buffer.  if error is not SUCCESS, the payload does not
   * contain valid data and the src pointer should not be used.
   */
  event void recvfrom(struct sockaddr_in6 *src, void *payload,
                      uint16_t len, struct ip_metadata *meta);

}
```

**Usage**

Each socket must be allocated using the generic component UdpSocketC.

For clients, no initialization is necessary; they may send to a destination without calling bind. The stack will allocate a unique ephemeral port number and send out the datagram.

Servers wishing to provide a service using a well-known port should call bind() on that port number before generating datagrams.

The simplest server is an echo service running on port 7.

Because of the buffer semantics, it is safe to call send directly from a receive event handler.

```
  event void Boot.booted() {
    call Echo.bind(7);
  }

  event void Echo.recvfrom(struct sockaddr_in6 *from, void *data,
                           uint16_t len, struct ip_metadata *meta) {
    call Echo.sendto(from, data, len);
  }
```

The wiring is as follows.

```
  components new UdpSocketC();
  UDPEchoP.Echo -> UdpSocketC;
```

## UDP-based Shell

UDPShell is a simple text-based command processor which comes with blip. It is an optional, although convenient way of adding interactive debugging commands to a mote. By default, the shell contains only a few simple commands: help, echo, uptime, ping, and ident. It is designed to be very easy to extend by adding your own commands.

To include just the basic shell, include the UDPShellC component in your application. To augment the shell with a new shell command, use the generic component ShellCommandC. Suppose we want to implement `expr`, a simple arithmetic evaluator. First, bind the 'expr' command string in your application configuration.

```
configuration App {} implementation {
  components UDPShellC, AppImplP;
  components new ShellCommandC("expr") as Expr;
  AppImplP.Expr -> Expr;
}
```

Within AppImplP, you must implement the `ShellCommand` interface. The interface has only one event, 'eval' which has the same prototype as `main()` in a typical c program.

```
event char *Expr.eval(int argc, char **argv) {
  char *ret = call Expr.getBuffer(15);
  if (ret != NULL) {
     strncpy(ret, "Hello, World!\n", 15);
```

```
  }
  return ret;
}
```

If expr returns a non-null value, it must be a null-terminated string which will be echoed back to a connected client. The buffer returned must obviously not be allocated on the stack. The shell component maintains a single buffer which components can use to print their reply to; it can be requested with a call to `Expr.getBuffer(uint16_t len)`. More examples of code using this interface are available within the stack; see `tos/lib/net/blip/shell/FlashShell[CP].nc` and `tos/lib/net/blip/nwprog/NWProg[CP].nc`

To use the shell, connect to the mote with the shell on UDP port 2000. You can use `nc6` (netcat) to do this interactively. Since the shell exports a number of builtins and can create a dynamically generated help string listing all available commands. For instance, if the mote has address `fec0::1`, we could try:

```
$ nc6 -u fec0::1 2000
help
sdsh-0.9        builtins: [help, echo, ping6, uptime, ident]
                [expr]
expr
Hello, World!
ident
        [app: TCPEchoC]
        [user: stevedh]
        [host: rabbit]
        [time: 0x4aaee400L]
```

In this example, the commands and reply from the mote are intermingled. Try out the other commands: use `ping6` the link-local multicast address (`ff02::1`), which should include at least one node (your upstream parent)!

## TCP

**WARNING: the TCP stack is still experimental, and may not provide the performance or reliability you are accustomed to.**

TCP is the standard Internet protocol for reliable, in-order delivery of data across the network. Although inefficient, its ubiquity makes it impossible to ignore; thus, blip provides a very simple TCP stack. TCP is considerably more complicated then UDP, and requires careful use in the embedded setting to prevent resource exhaustion. It is essential that one understand the BSD sockets API; this brief README does not cover many details.

For memory-constrained operation, blip's TCP does not do any receive-side buffering. Instead, it will immediately dispatch new, in-order data to the application and otherwise drop the segment. Blip does provide send-buffering so that it can automatically retransmit missing segments; this buffer may be of any size and is provided by the application.

The TCP interface is located in `$TOSROOT/tos/lib/net/blip/interfaces/Tcp.nc`. For the most part, it should be familiar. Since the application is responsible for buffering, both `accept()` and `connect()` require the implementer to include a buffer for the stack's use. Once passed to the stack, the buffer is reserved until a `closed()` event is signaled on that socket.

A few of the most important caveats:

- there is no `listen()`. calling `bind()` on a socket also begins to listen.
- there is no way to `accept()` multiple sockets like you can in Unix. More precisely, the code would support it but then there would be dynamic allocation since you have to allocate a new socket struct on the fly.
- As a result of these, if the socket is closed, you have to call `bind()` if you want to continue listening.
- You'll need to carefully manage buffer size by hand if you want to be sure of correct operation. Make sure you check return codes from `send()` since it will fail if there is not enough local buffer for the entire request.

### Example

```
configuration {
  components new TcpSocketC() as TcpEcho;
  TCPEchoP.TcpEcho -> TcpEcho;
}

module {} implementation {
  // allocate a send buffer
  char tcp_buf[150];

  // accept connections from anyone.  no need to save the endpoint,
  // but this is the only time its available (add an API call?)
  event bool TcpEcho.accept(struct sockaddr_in6 *from,
```

```
                            void **tx_buf, int *tx_buf_len) {
    *tx_buf = tcp_buf;
    *tx_buf_len = 150;
    // indicates we are accepting the connection
    return TRUE;
  }

  event void TcpEcho.connectDone(error_t e) {}

  // just echo the data back.
  event void TcpEcho.recv(void *payload, uint16_t len) {
    call TcpEcho.send(payload,len);
  }

  // rebind to accept other connections.
  event void TcpEcho.closed(error_t e) {
    call Leds.led0Toggle();
    call TcpEcho.bind(7);
  }
```

#### TCPEcho

`TCPEcho` is a sample application which comes with TinyOS which demonstrates using the TCP stack to build a very simple RESTful web service. It is located in `apps/TCPEcho` and can be installed using the same process as other blip applications: use `make <target> blip` to build the application.

Once it is installed on a mote, it provides the following services. Suppose it has been installed with ID 1 on the default (site-local) subnet.

- UDP echo on port 7
  - try `` `nc6 -u fec0::1 7` `` and typing a few characters.
- TCP echo on port 7
  - try `` `nc6 fec0::1 7` ``. This will look the same, but uses a reliable TCP transport. If you type too quickly, you will see gaps in the stream as various buffers overflow!
- RESTful web service on port 80
  - in a web browser, visit `http://[fec0::1]/read/leds`
  - the brackets in the URI are the notation for directly entering IPv6 addresses.

This application includes a components `HttpdP.nc` which may be a useful component for building your own RESTful services. Using an event-driven TCP stack takes some practice!

### Network Programming

`nwprog` is a method of over-the-air programming. It uses much of the machinery Deluge has developed, like the boot loader and flash layout, but substitutes a simpler transport using UDP for Deluge's dissemination algorithm. This means that it is point-to-point, and not incredibly appropriate for reprogramming an entire network all at once. `nwprog` differs from Deluge in several important regards:

- no dissemination
- no base station or serial port for injection

The application is very simple: flash is formatted into several volumes (a golden image and three application volumes), which are used to store application images. Flash management, boot loading, and image formatting are all provided by Deluge.

#### Usage

Build your application with `nwprog` support by including a line in your application Makefile, and include the IPDispatchC component. The UDP shell will be automatically included; it is necessary to interact with the stored images. Put this somewhere in your application Makefile:

```
BOOTLOADER=tosboot
```

Also, it is necessary to include a volumes xml file for your flash chip; an example for the stm25p can be found in `apps/UDPEcho/volumes-stm25p.xml`. See TEP103 (http://www.tinyos.net/tinyos-2.x/doc/txt/tep103.txt) for more information about this file.

After setting up your source, built the tosboot bootloader for your platform by going to `tinyos-2.x/tos/lib/tosboot` and typing `make <platform>`. Then just build and install your application like usual. If networking is working, you should have no problem following the rest of the instructions.

Interactions with the motes happen using the 'nwprog' tool in a shell. Connect to the shell with `nc6 -u fec0::1 2000`. If you have included network programming, the nwprog tool will be available. It has three commands:

- `nwprog list`: examine the flash and print out information on volumes containing images believed to be valid
- `nwprog reboot`: reboot into the same image
- `nwprog boot N`: reboot, and flash the mote with the image stored in volume N

In order to upload new images, use the tos-nwprog tool, located in $TOSROOT/tools/tinyos/misc. This tool provides minimal functionality; only erasing and uploading are supported.

- `./tos-nwprog -e 0 fec0::65`: erase image 0 from the mote at the given IP address.
- `./tos-nwprog -u 0 -f tos_image.xml fec0::65`: upload the image in tos_image.xml to volume 0 on the mote at the IP address. This will erase the volume before uploading it.

tos-nwprog provides several other features: `tos-nwprog --help` will print information about them.

To integrate with your own application, there are several internal interfaces which can be used to examine the flash. Looking at the example code in UDPShellP component is the best way of finding out about these.

# Further Reading

- blip-centered mailing list (https://lists.eecs.berkeley.edu/sympa/info/blip-users)
- How to cross compile the routing driver (http://smote.cs.berkeley.edu:8000/tracenv/wiki/blip/CrossCompiling)
- History of blip releases (http://smote.cs.berkeley.edu:8000/tracenv/wiki/blip)
- Crossbow BLIP tutorial (http://webcache.googleusercontent.com/search?q=cache:tuXrmKKWsg0J:blog.memsic.com/2009/07/in-a-blip-pervasive-ip-has-arrived.html+In+a+BLIP+,+pervasive+IP+has+arrived.&cd=1&hl=en&ct=clnk&gl=us&client=firefox-a)
- David Culler: TinyOS Meets IP (http://tinyos.stanford.edu/ttx/2007/viewgraphs/standards-ip.pdf)

*This file is licensed under the Creative Commons Attribution ShareAlike 3.0 (http://creativecommons.org/licenses/by-sa/3.0/) License. In short: you are free to share and make derivative works of the file under the conditions that you appropriately attribute it, and that you distribute it only under a license identical to this one. Official license (http://creativecommons.org/licenses/by-sa/3.0/)*

Error creating thumbnail: convert: no decode delegate for this image format `/tmp/magick-iItstrk' @ error/constitute.c/ReadImage/532.
convert: missing an image filename `/tmp/transform_b73ff6a704b3-1.png' @ error/convert.c/ConvertImageCommand/3011.

```
Error creating thumbnail: convert: no decode delegate for this image format `/tmp/magick-8KGTbY89xpD' @ error/constitute.c/ReadImage/532.
convert: missing an image filename `/tmp/transform_245e7087f04-1.png' @ error/convert.c/ConvertImageCommand/3011.
```

Retrieved from "http://tinyos.stanford.edu/tinyos-wiki/index.php?title=BLIP_Tutorial&oldid=5642"
Category: Tutorials

---

- This page was last modified on 24 October 2011, at 13:59.
- This page has been accessed 111,114 times.