© 2012 IEEE. Downloaded from NewAE.com. Reproduction, Republishing, or Re-Use Prohibited. Contact IEEE For Details.

FlexibleIP (FIP): IPv6 Stack for Experimental Work on Low-Power Wireless Networks

Colin O'Flynn
Electrical & Computer Engineering Department
Dalhousie University
Halifax, Canada
coflynn@newae.com

***Abstract*—IPv6 is often deployed in wireless sensor networks, and the advantages of such a deployment are well reported. For researchers which wish to experiment with new protocols or cross-layer design, there is often a choice of which IPv6 stack to base their work on. The FlexibleIP (FIP) stack presented here is an open-source design which makes documentation, code readability, and ease of modification a priority. In addition FIP provides compile-time options to support a wide variety of devices; it can fit into an 8-bit microcontroller, but can still use features typical when targeting larger microcontroller such as dynamic memory allocation for buffering many packets. The stack also offers excellent support for cross-layer design and hardware acceleration, two areas of special interest for low-power design. Conformance testing of the stack assures it adheres to relevant standards, and static analysis of the code is also performed. In addition to presenting this work, the paper highlights some design criteria that any IPv6 stack targeting low-power wireless networks should consider.**

Keywords-IPv6 Stack; 6LoWPAN; IoT; cross-layer design

I. INTRODUCTION

When designing a low-power wireless sensor network, the choice to use IPv6 provides several advantages. IPv6 on the network means true 'end-to-end' connectivity; this can greatly simplify integrating the sensor network into a larger architecture.

Potentially IPv6 packets may be too large to fit inside a single wireless packet; IEEE 802.15.4 has a maximum packet size of 127 bytes, significantly smaller than the minimum transmission unit (MTU) of 1280 bytes required by IPv6. In this example the 6LoWPAN protocol is used to provide both fragmentation and a degree of data compression on the wireless link, efficiently fitting the larger IPv6 packets into IEEE 802.15.4 payloads[1][2].

Most devices which will use this low-power link have constrained microcontrollers. The degree of constraint depends on the exact system – an energy scavenging device may only have a small 8-bit microcontroller, but a device with either mains power or a larger battery may have a 32-bit microcontroller. A typical network consists of a mix of devices; it is this usage scenario that the open source FlexibleIP (FIP) stack aims to fill[3]. FIP provides a variety of compile-time options to tune the stack to make the best use of the device, for example a small device uses a static storage buffer that deals with a single packet at a time. A larger device can use dynamically allocated packets, and deal with a queue of packets coming from multiple interfaces. In addition the code contains extensive static and dynamic verification features to reduce bugs; something often missing in academic projects.

Figure 1. FIP has passed IPv6-Ready Logo Phase-1 Testing for Hosts, demonstrating it conforms to relevant RFCs[4].

This paper will first introduce several other IPv6 stacks, and detail why FIP provides a useful alternative to them. From there the general design philosophy will be introduced, the host interface, and an example of running FIP on a computer will be given. From there an modifying the stack will be discussed, along with tools for code verification and debugging. Finally a few examples of code sizes for both 8-bit and 32-bit architectures are provided. It should be noted the code size is highly dependent on compile-time configuration – table 1 gives some of the options FIP have that affect code size.

TABLE I. COMPILE TIME OPTIONS AFFECTING CODE SIZE

Option	Allowed Values
Link-Layer Address Length	Fixed, Variable
Packet Buffer Type	Statically allocated single buffer, Statically allocated multi-buffer, Dynamically allocated (malloc)
ICMPv6 Error Message Support	Enabled, Disabled
IPv6 Fragmentation Support	Enabled, Disabled
UDP Support	Enabled, Disabled
6LoWPAN Support	Enabled, Disabled
Maximum number interfaces	1 – 255[a]
Addresses per interface	1 – 255[a]
Number of default routers, neighbour entries, routing entries, etc	1 – 255[a]

a. Can increase upper limit easily at compile time as well

© 2012 IEEE. Downloaded from NewAE.com. Reproduction, Republishing, or Re-Use Prohibited. Contact IEEE For Details.

II. State of the Art

Several stacks target very small devices – Contiki's uIPv6 stack being the first major player in this area[5], although a number of other stacks now exist targeting similar devices as detailed in [6][7]. In order to get the lowest code size these stacks may use a code style which is more difficult to understand, and it is also difficult to extend in ways not originally intended. FIP values flexibility and readability over absolute minimum code size – FIP will *not* result in the smallest implementation. It will be demonstrated in section 10 that the cost of devices using FIP is comparable to the cost of using one of the smaller stacks.

Stacks originally designed for larger devices, such as the IPv6 stack that is part of the Linux kernel, are difficult to strip down for smaller devices. In addition the stacks may require changes for use in 6LoWPAN environments. Cross-layer design is often using for 6LoWPAN protocols, but in the Linux stack such cross-layer programming is often discouraged, and thus the APIs for accessing lower-layer data from higher layers may not exist.

FIP provides a stack that can be adjusted to fit both small and large devices, which allows a diverse network to use a single IPv6 stack. This can speed development since there is no learning curve when switching between devices.

One issue unique to IEEE 802.15.4 networks is that a single interface may have two addresses: this is because IEEE 802.15.4 contains both 16-bit and 64-bit addresses, and either may be used to identify the interface. Many IPv6 stacks are not written with consideration for this, where FIP natively supports an arbitrary number and length of link-layer network addresses.

IPv6 stacks often come tightly integrated within a RTOS. Contiki's uIPv6 comes tightly integrated with the Contiki RTOS and build system, for example. Users which already have code working on another RTOS or build system may see this as a disadvantage. As FIP makes little assumptions about the services available from the host, it is easier to integrate into a different build systems or RTOS. The entire code is standard ANSI C, which almost all embedded systems support.

III. Design Philosophy and Packet Buffering

FIP works on the idea that the stack is always operating on a single "active packet buffer". This design allows very constrained devices to only have a single active packet. More complex systems can easily store thousands of packets, switching the active one based on a scheduler. For processing IP data the resulting architecture becomes extremely efficient: the metadata surrounding the IP data itself always follows the packet, and can be used as the basis for deciding when to advance that packet through different stages. Each packet contains a 'state' flag for example, which can be used for the stack to keep track of things such as waiting for address resolution to finish, or waiting for IPv6 layer fragmentation to finish. This is shown graphically in fig. 2.

The choice of a single 'active' packet buffer compared to always passing a pointer to the buffer being operated on reduces needlessly passing the same pointer around. Indeed when the option to use only a single 'active' buffer is enabled, the code automatically optimizes these into a single statically allocated buffer, which has significant code saving in small devices such as the Atmel AVR.

The active packet buffer can contain several individual buffers linked together. Using a single flat buffer would mean a costly move operation whenever an IPv6 extension header needs to be inserted or removed; this is of particular importance when routing protocols need to modify a header at each hop, or when routing between two different networks. In addition the low-level network interface writes the received data directly into a buffer provided by FIP, rather than into an intermediate buffer which is then copied over.

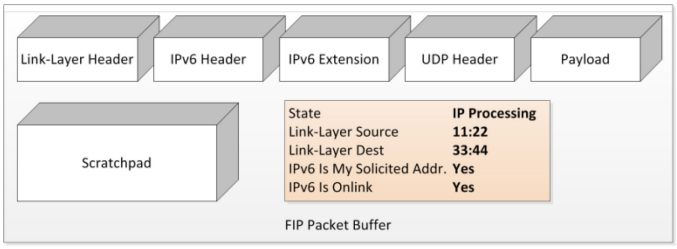

Figure 2. Each packet received or generated is put into a 'packet buffer', which includes data buffers along with metadata that follows the packet.

The metadata surrounding the packet includes information not directly in the packet, but that will be needed later in the processing of that packet. This metadata can include data which is encoded in the packet: for example the metadata can mark a packet as being the node's solicited-node multicast address, this is checked when the packet is first received, since if the packet is addressed to this node is it handled differently. Later when ND code needs to check the packet is addressed to a solicited node address, it doesn't need to do the entire comparison again, since the results of the previous comparison were stashed in the metadata. The metadata also solves the problem of how to manage interaction between different layers. In low-power networks the idea of 'cross-layer' design is popular, and 6LoWPAN for example is often seen as a 'layer 2.5' solution since it needs information about both the lower (physical) addresses along with IPv6 address information. All relevant information is stored in the metadata, and can be natively accessed throughout processing of the packet by different layers.

A 'scratchpad' is also provided, which is a block of bytes used for a state-specific purpose. During 6LoWPAN reception, for example, the code needs to track information such as the current fragment number and identification. While the packet buffer is in the '6LoWPAN Reception' state, that scratchpad is reserved for use by the 6LoWPAN layer. Ideally the 6LoWPAN layer would have dynamically allocated memory which it would then free once complete, but this is difficult for smaller devices which do not support dynamic memory allocation (malloc). The scratchpad thus can be thought of a simple way of making efficient use of memory without requiring malloc support. In addition the scratchpad avoids responsibility of each layer needing to `malloc()` and `free()` their own personal memory block, which could lead to fragmented memory or a leak.

© 2012 IEEE. Downloaded from NewAE.com. Reproduction, Republishing, or Re-Use Prohibited. Contact IEEE For Details.

IV. Host Interface

The minimum required host interface is shown in listing 1. These are functions which need to be defined for FIP to operate; they deal primarily with getting the time, and a single function to seed the random number generator.

IPv6 stacks have many timed events such as expiring addresses, neighbors, routes; timing retransmissions; and keeping logs. To simplify integration FIP does not rely on a host-provided event queue, instead timing these events by periodically polling for their expiration. The resulting jitter due to polling is well within acceptable limits for IPv6 as the timing is not critical; many of the events already have artificially added jitter to avoid synchronizing events between two nodes.

```
uint32_t fip_hostInterface_getSeconds32 (void)
uint16_t fip_hostInterface_getSeconds16 (void)
uint32_t fip_hostInterface_getMiliSeconds32 (void)
uint16_t fip_hostInterface_getMiliSeconds16 (void)
uint32_t fip_hostInterface_getRandom32 (void)
```

Listing 1. Functions that FIP assumes the host provides.

A. Generic Network Interface

The network interface is responsible for passing IPv6 packets between FIP and the physical medium. The exact interface varies depending on the medium, as this code is responsible for tasks such as removing or adding headers and validating any other low-level protocol information.

In general when the network interface receives a packet, it will call FIP to allocate a new packet. To minimize double-buffering, the allocation can be done early on in the reception. Now data is copied into the buffer provided by FIP, and then information such as the link-layer addresses is written into the metadata surrounding the packet. Once the network interface is done with the packet, the state of the packet is set to 'Ready for IP Processing', which tells FIP that a full frame has been received. If the network interface requires additional time, such as the case when receiving many link-layer fragments, it can simply set the state of the packet as 'Reception Ongoing'. The IP stack will leave the buffer alone until the network driver finishes with it in subsequent calls.

When FIP wishes to send a packet out, it simply sets the state to indicate it is ready for transmission. The network driver will detect this state change, and send the packet. The metadata can provide 'hints' to the network driver about this transmission process – for example if there are several possible source addresses, as is the case for IEEE 802.15.4, the metadata can specify the preferred link-layer source address to use. If nothing is specified, the network driver uses a default. For cross-layer design, this means application layer programs can change the metadata and select a preferred source address.

B. 6LoWPAN Interface

FIP provides a 6LoWPAN layer, which has a different interface than the generic network interface. Where the generic network interface passes IPv6 frames, the 6LoWPAN interface expects data from IEEE 802.15.4 frames. When a frame is received, the function `fip_sixlowpan_llInput()` is called, which is passed the addresses (either 16-bit or 64-bit) and payload. The IEEE 802.15.4 MAC is responsible for passing this data to 6LoWPAN, which might include decrypting frames in networks using security.

Sending a frame is done through four function calls. First FIP will call `fix_hostInterface_sixGetPayloadLength()` which tells the host the address types (16-bit or 64-bit) being used, and the host responds with the number of bytes of payload which could fit in the resulting IEEE 802.15.4 packet – this feedback is critical, since depending on what security is being applied by the host the number of bytes will vary. This interface also simplifies expanding the 6LoWPAN code to work on layers besides IEEE 802.15.4. FIP will then call `fip_hostInterface_sixSendSetup()` which tells the host the actual addresses, and then pass the host a block of data with `fip_hostInterface_sixSendCopyPayload()`. Once data copying is finished it will call the function `fip_hostInterface_sixSendFinish()`. This tells the host the packet can be sent out. Many IEEE 802.15.4 MACs will not transmit a packet immediately, but instead queue it. The decision to use several callbacks was made because it allows FIP to ready several packets for transmission, and not need to wait until the MAC is actually finished with the packet before preparing the next one.

V. Simulator Environment

For development, debugging, and testing FIP can be run directly on a host computer. Debugging your design does not require an In-Circuit Emulator (ICE) or special software; instead the familiar PC development is used. A simulated IEEE 802.15.4 interface is provided, which connects several instances of FIP running on the computer. Wireshark can directly sniff this, as shown in fig. 3. Interfacing FIP to a simulator such as ns2 would be trivial if more control of the IEEE 802.15.4 packet transmission is desired.

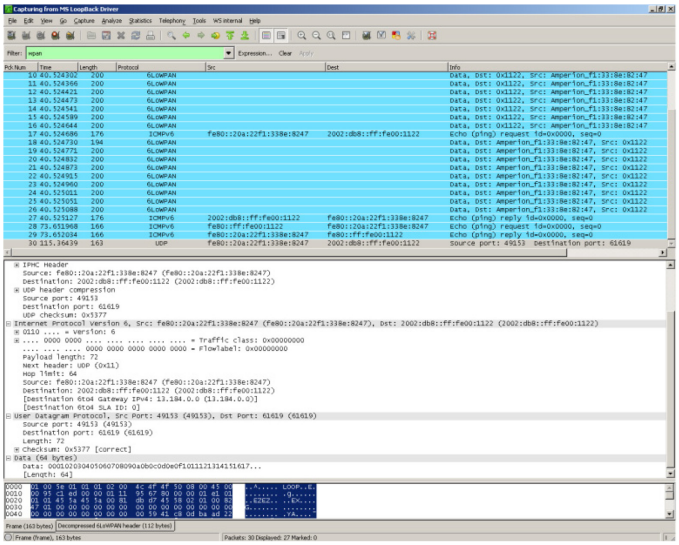

Figure 3. The simulator uses the loopback interface to send IEEE 802.15.4 packets between the different instances of FIP. If wireshark is attached to the loopback interface, it can see these packets and decode them as it would an Over-The-Air sniffer.

The simulated FIP environment can also take control of the hardware Ethernet port on the computer, allowing the entire system to be used in a real network. Adding support for another

© 2012 IEEE. Downloaded from NewAE.com. Reproduction, Republishing, or Re-Use Prohibited. Contact IEEE For Details.

hardware interface, such as an IEEE 802.15.4 USB radio, would be trivial.

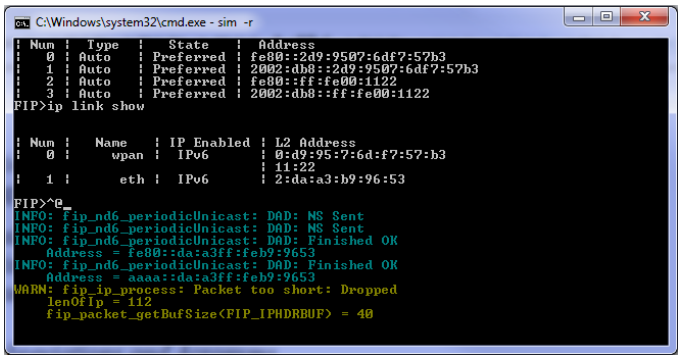

Figure 4. Running FIP on the host 'simulator' provides a console-style interface. The top half of the screen is the console into which commands are typed and FIP responds, and the bottom half is trace messages forming the built-in debug channel. See section 8 for details.

VI. Addition of New Features

Of most interest to researchers is the ability to add new features, protocols, or algorithms. In this capacity FIP excels, as it was specifically designed to allow the addition of modules without requiring 'hacks' to the underlying code.

As an example consider the 6LoWPAN-ND module [8]. Implementing 6LoWPAN-ND on a normal IPv6 stack is difficult, since it requires many changes to features such as neighbor discovery (ND) which are core to the IPv6 stack. These changes must only operate on interfaces which are actually using 6LoWPAN-ND, as a node may have both regular Ethernet interfaces and 6LoWPAN interfaces.

As FIP operates on the state of the active packet, adding support for 6LoWPAN-ND requires a modification to the processing of states which involve ND, such as the "Address Resolution Required" state. In this case if the interface on which address resolution is required supports 6LoWPAN-ND, flow can be passed to the 6LoWPAN-ND module instead of normal ND. In addition a call to 6LoWPAN-ND is added into the 'maintenance' task, which changes how neighbor table entries are maintained.

Because information is stored in metadata alongside the packet, it is very clean to implement protocols which require information from other layers. The cross-layer approach is popular in protocols attempting to reduce duplication and wastage, so natively supporting them without requiring 'hacks' to the code is a huge advantage.

VII. Verification

Simply testing a device using the application being developed is insufficient to consider it reliable. FIP itself provides two main levels of verification: IPv6 standard conformance and interoperability testing using the IPv6 Ready Logo program, and continuous static analysis of the codebase using Gimpel PC-Lint.

A. IPv6 Ready Logo

The IPv6 Ready Logo testing consists of two main sections: interoperability testing, and conformance testing[9]. The interoperability test puts FIP in a network with devices and stacks from other vendors, and ensures it works with all the devices. The conformance testing meticulously tests FIP against relevant RFCs, to ensure it is operating correctly according to those standards.

The conformance testing provides an extremely thorough example of the level at which FIP is operating. Stacks which have not passed IPv6 ready testing make no guarantee about how closely they actually follow relevant RFCs. Table 2 contains the RFCs tested, number of tests applied, and a brief description of what those tests encompass.

FIP is currently considered IPv6-Ready Logo Phase-1 approved for host devices as shown in fig. 1[4].

TABLE II. IPv6 Ready Comformance Tests

Specification	Tests	Description
RFC2460 – IPv6	29	Handling of IPv6 header fields, fragment reasssembly
RFC4861 – Neighbor Discovery for IPv6	116	Handling of ND fields, timeouts, address resolution queue, handling ND messages
RFC4862 – Stateless Address Autoconf	43	Address timeouts, prefix, address creation and duplicate detection
RFC4443 – ICMPv6	8	Echo Request / Response

B. Static Analysis

The use of static analysis is well known as one tool which can reduce bugs to achieve high-quality software [10]. Gimpel PC-Lint was chosen as the static analysis tool, and scripts to easily run this tool against the FIP source are included in the FIP distribution. Since development has always included this static analysis, it is trivial for new code to be checked without needing to spend time working through the many false positives that typically occur the first time trying to run Lint.

A project does not need to use static analysis from the beginning to unlock these results, as any code-base can be analyzed at a later time. The number of 'false' hits in such code may be daunting though, and results in the static analysis being pushed aside due to the difficulty of going through every reported problem.

Static analysis is also useful when compiling the unchanged code on a different architecture – for example to catch that the width of a variable changed from an expected size, because code did not use one of the `stdint.h` types. Such bugs are difficult to debug, since often they only occur after that specific variable happens to overflow, which may take days or weeks of operation. Static analysis provides confidence that a new architecture will not have such problems.

VIII. Debug & Console

A. Debugging & Trace Statements

Debugging on small low-powered devices is often difficult. For example logging debug messages can be helpful in finding problems. But printing and logging these messages can take huge amounts of code space and time – the serial port, a frequent choice for debug messages, is relatively slow compared to normal execution speed of microcontrollers. Thus

© 2012 IEEE. Downloaded from NewAE.com. Reproduction, Republishing, or Re-Use Prohibited. Contact IEEE For Details.

it can easily become overloaded with messages in a busy network.

FIP provides a debug interface which can be configured to either print normal debug strings, or instead print short binary tokens which refer to pre-defined strings. These pre-defined strings are stored in a header file which can be built into both the FIP integrated target or an application on the computer which is decoding the tokens. Thus it remains trivial to update and maintain this file – critical if one expects it to be used in the future.

In addition this debug interface supports adding the value of variables to the output, which will automatically be formatted to appear as part of the previous debug statement. The end objective of this system is to provide a method which is easier for programmers to use than a `printf()` – as an example take listing 2 and 3. The `TRACE()` statement takes three arguments: a level, trigger, and action. In this case the `DEBUG` level means an event which occurs normally, and would only be printed when looking for the most detail about code execution. The `ROUTEFAILED` is the trigger, meaning the code has detected routing to a node has failed. Finally as a result of this 'something' has been `DROPPED` – in this case the next line prints the IPv6 address of a router which has been dropped from the default route table. The macros will print the name of the function in which the event occurred, which both provides some context for what is happening, and also makes it easy to find the exact spot in the code these statements were called.

```
TRACE(DEBUG, ROUTEFAILED, DROPPED);
TRACE_ADDIP6ADDR(DEBUG, "router", ipaddr);
```

Listing 2. C Source code for adding a debug statement.

```
DEBUG: fip_if6_addUpdateDefaultRouter: Route Failed:
Dropped.
        Router = fe80::ff:fe00:3344
```

Listing 3. Resulting human-readable output from executing the debug statements in listing 2.

B. Console

While working with nodes, it can be useful to have a method of manually overriding items such as addresses, routing tables, and neighbor tables. FIP again provides a comprehensive console to perform these activities, table 3 shows an example of the commands supported by this console.

TABLE III. CONSOLE COMMANDS SUPPORTED

Command	Operation
Trace	Set trace level: setting 'debug' for example prints all messages, setting 'warn' only prints warnings or errors.
Queue	Show packet queue – for example packets waiting for additional IPv6 fragments.
Link	Show and modify link attributes such as address, name, and if routing is enabled.
Addr	Show and modify IP addresses assigned to each interface.
Pref	Show and modify the prefix table.
Neigh	Show the neighbour table.
Route	Show and modify the routing table.
Context	Show and modify 6LoWPAN compression contexts.
ping6	Perform an IPv6 ping.
Udp	Simple netcat-style program which can send UDP data to an arbitrary port on a remote host, or listen locally to UDP ports.

IX. DOCUMENTATION

This paper has provided a brief overview of FIP; more detailed documentation is maintained through the Doxygen system in the source code. Not only does this include API documentation, but also discussions of architecture, how to run static analysis, and coding style and conventions used throughout the project. This documentation is available from the FIP website [3].

X. IMPLEMENTATION EXAMPLES

As FIP provides a highly portable IPv6 stack, the code-base does not include CPU-specific code or drivers. It can easily be compiled onto any existing code which provides the required network interface, such as a vendor-supplied MAC.

This section includes some examples of code size for various platforms. All these code numbers have IPv6 fragmentation and ICMPv6 error messages disabled; these features are required for IPv6 ready conformance testing, but most small IPv6 stacks do not implement them. Detailed comparison between stacks is difficult not only because of feature differences, but because stacks may be configured with a different number of buffers, neighbor table entries, etc. Code size reported here is only of FIP code and does not include any required drivers or other host code.

TABLE IV. CODE SIZE EXAMPLES FOR SOME ARCHITECTURES

Architecture	Interfaces	Buffer	ROM (bytes)	RAM (bytes)
Atmel AVR 8-bit	6LoWPAN	Static	32 768	3 386
Atmel AVR 8-bit	Eth+6Low	Static	33 001	3 946
Atmel SAM3 Cortex-M3	6LoWPAN	Static	25 197	2 972
Atmel SAM3 Cortex-M3	6LoWPAN	Dynamic	28 129	1 691 [a]
Atmel SAM3 Cortex-M3	Eth+6Low	Dynamic	28 193	2 248 [a]
Xilinx Microblaze	6LoWPAN	Dynamic	45 074	2 004 [a]
Xilinx Microblaze	Eth+6Low	Dynamic	45 150	2 688 [a]

a. Does not include dynamic RAM required for holding IPv6 packets

It should be noted that FIP primarily targets devices containing > 64K of FLASH (ROM) and > 8K or SRAM. This design decision is driven by market economics: the cheapest solution for low-power wireless chips is often a complete System-on-a-Chip (SoC) that integrates both radio and microcontroller. These SoC devices tend to be either high-end 8-bit microcontrollers or 32-bit ARM microcontrollers. As an example Atmel's ATMega128RFA1, a microcontroller and radio SoC, has 128K FLASH and 16K SRAM for $5.28[1]. Assuming we had an optimized stack which fit within 64K FLASH and 4K SRAM, using a separate radio and

© 2012 IEEE. Downloaded from NewAE.com. Reproduction, Republishing, or Re-Use Prohibited. Contact IEEE For Details.

microcontroller would mean an AT86RF231-ZU radio which costs $2.64[1] combined with an AtMega6450A-AU microcontroller which costs $3.50[1], for a total cost of $6.14. As prices continue to fall there is little incentive to arbitrarily optimize code at the expense of flexibility or readability.

The code contains macros for a variety of tasks which could be performed by special hardware peripherals or instructions. This includes for example memory copies, compares, and table lookup. As FIP can be compiled for the Xilinx Microblaze core, this would be an ideal platform for experimenting with hardware-assisted IPv6 stacks [11].

XI. Performance Measures

A simple performance comparison will be presented between FIP and the IPv6 stack in Contiki[5]. Table V shows the number of clock cycles required for sending a 6-byte UDP message over a 6LoWPAN network when implemented on an ATMega128RFA1 8-bit microcontroller. Contiki takes fewer total cycles (9077 vs. 11103), which is expected due to the highly optimized design of Contiki. At the 16 MHz clock speed of the test platform this corresponds to 127 µS difference, or about the time to transmit 4 bytes of IEEE 802.15.4-2006 traffic.

TABLE V. Performance Comparison

Layer	Contiki uIPv6	FIP
UDP	304	5146
IPv6	5307	2571
6LoWPAN	3466	3386

XII. Conclusions and Future Work

The FlexibleIP (FIP) Stack provides a useful reference and base for experimenters. Notably absent from the implementation is TCP code, which also prevents an HTTP implementation. Future work will add an optional TCP module, or it may be possible to call existing TCP code from FIP.

The UDP implementation allows many higher-layer protocols used in constrained networks such as CoAP[12]. The goal of FIP is to provide a flexible base, so it is not expected to implement such higher-layer protocols. Publically available implementations could easily call FIP using the provided API.

By concentrating on documentation, code readability, and flexibility FIP aims to provide the most useful reference IPv6 stack for experimenters. It is especially suited to work in cross-layer designs, hardware acceleration of IPv6 stacks, and networks experimenting with heterogeneous device sizes. In addition FIP provides a simple simulator that natively runs the IPv6 stack on a PC, which can further be interfaced to a network-layer simulator such as ns2.

FIP is open-source with a permissive license, allowing use in projects which may become commercialized.

[1] Source: Digikey in USD, March 31, 2012 in quantities of 100

References

[1] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "RFC4944: Transmission of IPv6 Packets over IEEE 802.15.4 Networks," 2007 [Online]. Available: http://tools.ietf.org/html/rfc4944.

[2] J. Hui and P. Thubert, "RFC6282: Compression Format for IPv6 Datagrams," 2011 [Online]. Available: http://tools.ietf.org/html/rfc6282.

[3] C. O'Flynn, "The Flexible IP Stack," 2011 [Online]. Available: www.newae.com/fip.

[4] IPv6 Forum, "IPv6 Ready Details for FIP," 2010 [Online]. Available: https://www.ipv6ready.org/db/index.php/public/logo/01-000567/.

[5] M. Durvy et al., "Making sensor networks IPv6 ready," *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pp. 421-422, 2008.

[6] J. J. P. C. Rodrigues and P. A. C. S. Neves, "A survey on IP-based wireless sensor network solutions," *International Journal of Communication Systems*, vol. 23, no. 8, pp. 963-981, 2010.

[7] C. Yibo et al., "6LoWPAN Stacks: A Survey," *Wireless Communications, Networking and Mobile Computing (WiCOM), 2011 7th International Conference on*, pp. 1-4, 2011.

[8] Z. Shelby, S. Chakrabarti, and E. Nordmark, "draft-ietf-6lowpan-nd-18: Neighbor Discovery Optimization for Low Power and Lossy Networks (6LoWPAN)," 2011 [Online]. Available: http://datatracker.ietf.org/doc/draft-ietf-6lowpan-nd/.

[9] A. Vallejo, J. Ruiz, J. Abella, A. Zaballos, and J. M. Selga, "State of the art of IPv6 conformance and interoperability testing," *Communications Magazine, IEEE*, vol. 45, no. 10, pp. 140-146, 2007.

[10] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *Software Engineering, IEEE Transactions on*, vol. 32, no. 4, pp. 240-253, 2006.

[11] P. T. Huang and W. Hwang, "A 65 nm 0.165 fJ/Bit/Search 256x144 TCAM Macro Design for IPv6 Lookup Tables," *Solid-State Circuits, IEEE Journal of*, no. 99, pp. 1-1, 2011.

[12] Z. Shelby, K. Hartke, and C. Bormann, "draft-ietf-core-coap-10: Constrained Application Protocol (CoAP)," 2012 [Online]. https://datatracker.ietf.org/doc/draft-ietf-core-coap/