Alexander Wachter, BSc

Design and Implementation of 6LoCAN, a 6Lo adaption layer for Controller Area Networks

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master´s degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing. Markus Schuss BSc
Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Eugen Brenner

Institute of Technical Informatics

Graz, February 2020

Affidavit

Date	Signature

Eidesstattliche Erklärung

Datum	Unterschrift

Abstract

Nowadays, there is an ongoing trend towards end-to-end IPv6 for constrained devices. This way, the devices can benefit from the vast amount of application and transport layer protocols, defined on top of the Internet Protocol. Examples for such application layer protocols are MQTT, CoAP, HTTP, or transport layer protocols such as UDP, TCP, and TLS. However, most devices on the Internet use Link-Layers that do not fit the needs of constrained devices like power consumption, price, or PCB footprint.

The Controller Area Network (CAN) is a very robust and simple bus. Lots of tiny microcontrollers have an integrated CAN controller that only needs an external transceiver to connect to a bus. This bus is usually used in the automotive and industrial domains. An example is the CANopen protocol, designed and used for automation. However, the protocols for the CAN bus serve a dedicated purpose and are not as flexible as the Internet Protocol.

Therefore, this work proposes 6LoCAN, an abstraction layer for the CAN bus, which combines the great flexibility of IPv6 with the benefits of the CAN bus. With 6LoCAN, it is possible to connect small microcontrollers to the Internet, with only little effort. Those devices can then use application layer protocols to communicate with devices that have Link-Layers of all kinds, like Wi-Fi, Ethernet, or Bluetooth, without the need for a gateway. Meanwhile, 6LoCAN has an IETF standard proposal and a reference implementation in the Zephyr Real-Time Operating System, as an outcome of this work.

Kurzfassung

Heutzutage erscheinen vermehrt elektonische Kleingeräte die über Ende-zu-Ende IPv6 kommunizieren. Auf diese Weise profitieren solche Geräte von den vielen Anwender- und Transport-Protokollen, die auf dem Internet Protokoll basieren. Beispiele für solche Anwender-Protokollen sind MQTT, CoAP, HTTP oder Transport-Protokolle wie UDP, TCP und TLS. Nichtsdestotrotz verwenden die meisten Geräte im Internet Verbindungsschichten, die den Anforderungen wie Energieverbrauch, Preis oder Flächenverbrauch auf der Leiterplatte von Kleingeräten nicht besonders gut erfüllen.

Controller Area Network (CAN) ist ein sehr einfacher und robuster Bus. Viele kleine und günstige Mikrokontroller haben einen integrierten CAN Modul. Diese benötigen nur noch einen Bus-Treiber Baustein um sich mit dem Bus zu verbinden. Der CAN Bus wird hauptsächlich im Automobil und Industrie Sektor verwendet. Ein Beispiel dafür ist das CANopen Protokoll. Nichtsdestotrotz sind die verwendeten Protokolle für eine dedizierte Aufgabe geschaffen und verfügen nicht über die Flexibilität des Internet Protokolls.

Deswegen stellt diese Arbeit 6LoCAN vor, einene 6Lo Abstraktionsschicht für den Controller Area Network Bus, welche die Flexibilität des Internet Protokolls mit den Vorteilen des CAN Bus verbindet. Mit 6LoCAN ist es möglich einfache Mikrokontroller, mit geringem Aufwand, mit dem Internet zu verbinden. Diese Geräte können wiederum über die Anwenderschicht-Protokolle mit geräten kommunizieren, die eventuell ganz andere Verbindungsschichten wie Ethernet, Wi-Fi oder Bluetooth verwenden. Für 6LoCAN existiert mittlerweile ein Vorschlag für einen Standard bei der IETF und eine referenz Implementierung im Zephyr Echtzeitbetriebsystem, welche im Zuge dieser Arbeit entstanden sind.

Contents

Abstract iii

1 Introduction 1

2 Background Knowledge 3
- 2.1 Controller Area Network . . . 3
  - 2.1.1 Wiring and Bus Access . . . 3
  - 2.1.2 CAN Frames . . . 6
- 2.2 ISO-TP Transport Protocol . . . 9
- 2.3 IPv6 . . . 12
  - 2.3.1 IPv6 Addresses . . . 13
  - 2.3.2 IPv6 Header . . . 18
  - 2.3.3 ICMPv6 . . . 19
  - 2.3.4 Neighbor Discovery Protocol . . . 20
  - 2.3.5 Stateless Address Autoconfiguration . . . 24
- 2.4 6lo . . . 26
  - 2.4.1 IP Header Compression . . . 27
- 2.5 Zephyr . . . 33
- 2.6 Zephyr Network Stack . . . 34

3 6LoCAN design 37
- 3.1 Addressing Schema . . . 38
  - 3.1.1 Unicast Address . . . 40
  - 3.1.2 Multicast Address . . . 40
  - 3.1.3 Address Generation . . . 41
  - 3.1.4 Link-Layer Duplicate Address Detection . . . 41
- 3.2 Stateless Address Autoconfiguration . . . 43
- 3.3 ISO-TP for 6LoCAN . . . 45
  - 3.3.1 Multicast . . . 45

3.3.2 Ethernet Border Translator 47

4 Implementation **51**
4.1 The Zephyr CAN API 51
4.1.1 Zephyr CAN frame 52
4.1.2 Zephyr CAN filter 52
4.1.3 Sending CAN frames 53
4.1.4 Receiving CAN frames 53
4.2 6LoCAN Implementation 54
4.2.1 6LoCAN Network Device Driver 59
4.2.2 6LoCAN Link-Layer 60

5 Evaluation **63**
5.0.1 Link-Layer Duplicate Address Detection 65
5.0.2 Ping 66

6 Conclusion **69**

List of Figures

2.1 CAN Physical Bit Transmission 4
2.2 CAN Wiring 4
2.3 Simplified CAN Transceiver 5
2.4 CAN Frame Format 6
2.5 CAN Base Frame 7
2.6 CAN Extended Frame 7
2.7 Bit Stuffing Example for 0x780 Identifier 8
2.8 Example Sequence of a Segmented Packet with a BS of three 10
2.9 Subnet Prefix of an IPv6 address 13
2.10 Link Local Unicast address 14
2.11 Multicast address 14
2.12 Global Unicast address 16
2.13 IPv6 Header 17
2.14 Internet Control Message Protocol Format 19
2.15 IPv6 Header Encapsulation 20
2.16 Neighbor Solicitation Message Example 21
2.17 Neighbor Advertisement Message Example 21
2.18 Router Solicitation Message Example 23
2.19 Router Advertisement Message Example 23
2.20 Stateless Address Autoconfiguration 25
2.21 6lo Adaption layer example 26
2.22 IPHC Frame Format 27
2.23 IPHC Format 28
2.24 Context ID Format 29
2.25 Next Header Compression for UDP Headers 31
2.26 Link-Local IPv6 an UDP header to IPHC example 32
2.27 Zephyr Network Stack 6LoCAN RX example 35

3.1 Address to Identifier Mapping 38

3.2 Unicast Node-Address to Identifier Mapping 40
3.3 Multicast-Group to Identifier Mapping 40
3.4 Link-Layer Duplicate Address Detection Request Frame . . 42
3.5 Link-Layer Duplicate Address Detection Response Frame . 43
3.6 Link-Layer Duplicate Address Detection 44
3.7 Example Multicast Sequence 46
3.8 Ethernet Border Translator Schematic 47
3.9 6LoCAN to Ethernet Address Translation 48
3.10 Ethernet to 6LoCAN Address Translation 49

4.1 Zephyr Network Stack 6LoCAN RX example 55
4.2 Zephyr Network Stack 6LoCAN TX example 58

5.1 Test Setup 64
5.2 Link-Layer Duplicate Address Detection Measurement . . . 65
5.3 UDP Transfer 128 byte, BS=0, STmin=0 68
5.4 UDP Transfer 128 byte, BS=8, STmin=0 68
5.5 UDP Transfer 128 byte, BS=8, STmin=5 68

1 Introduction

Nowadays, there is an ongoing trend towards end-to-end IPv6 for constrained devices. This trend is called the Internet Of Things (IoT). It means that many devices are connected, using internet technology. Some devices like smartphones or personal computers usually have a connection to the Internet out of the box, but more and more devices that usually don't have a connection to the Internet become connected devices. Examples are light-bulbs, fridges or building automation systems [17]. The advantage of using the Internet Protocol is that there are plenty of standardized application layer protocols like CoAP or MQTT, that can be used for those devices. "Project Connected Home over IP" [1], for example, is a project from global players like Apple, Google, Amazon, and many more, that is trying to establish a standardized interface for home application, based on the Internet Protocol. With this kind of interface, devices from different vendors and different communication mediums, like Wi-Fi, Ethernet, or Bluetooth Low Energy (BLE), can work together seamlessly.

The Eclipse Sparkplug working group [9] is currently working on defining a standard to use MQTT in the Industrial IoT. This initiative shows that there is a high demand for IP technologies in the industry.

There are lots of Link-Layer technologies already in use. They all have their domain-specific use-case. Ethernet or Wi-Fi are mostly used for high-speed data links suitable for the typical use case of a PC or smartphone. When using these technologies for tiny devices like a light-switch, problems like high costs, large PCB footprint, or high energy consumption may arise. The IoT, as we have it today, is mostly based on wireless technologies, but sometimes it is just not feasible to use a wireless link. Wireless links are prone to electromagnetic disturbances and have problems with range, especially when used in buildings with massive walls. The CAN-bus is a very robust and cheap bus that is widely used in the automotive and industrial automation (CANopen [2]) domain. It is also used for heating systems and thus already used in building automation today.

The combination of IPv6 and the CAN bus could be very useful to solve lots of challenges with technology we are already using today. With 6LoCAN, it is possible to connect devices with a wired bus to any other device that works with IPv6.

2 Background Knowledge

2.1 Controller Area Network

Controller Area Network (CAN) is a serial two-wire bus specified as Classical CAN in the Bosch CAN specification [5] and extended in the CAN FD specification [6]. One wire is the CAN High (CAN H), the other CAN Low (CAN L). The bus can either be in the recessive state, which is seen as a logical one, or in the dominant state, which is seen as logical zero. The so-called ”wired-AND” structure enforces a dominant bit to override a recessive bit. For transmitting a recessive bit, the bus is in the idle state, where both lines are at the same voltage level. For writing a dominant bit, the CAN H wire is tied to the positive voltage supply, and the CAN L wire is tied to a low level. Figure 2.1 shows this principle. CAN use a Non-Return-to-Zero Coding (NRZ), which means that the level at the bus is held for the entire bit time. A bit stuffing mechanism provides reliable synchronization for the time of the frame transmission. The idle level for both wires should be kept at a constant level about half the supply level. However, the physical layer is neither fully specified by the Bosch CAN specification [5] [6] nor the ISO CAN specification [15] and is left to the system integrator.

2.1.1 Wiring and Bus Access

For wiring, a two-wire twisted pair cable with an impedance of 120Ω is used. The topology is a line topology where stubs are allowed, but with a maximum length, depending on the bus speed. On the first and last node of the bus line, terminating resistors with a value 120Ω avoid reflections on the bus line. An example with two nodes is shown in Figure 2.2.

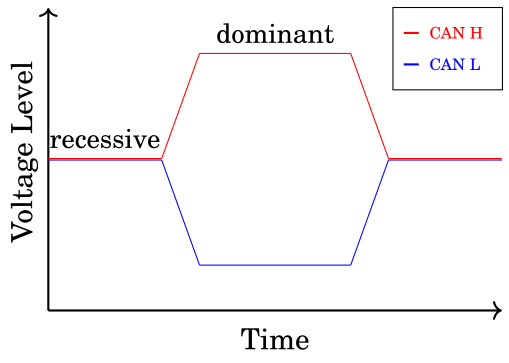


Figure 2.1: CAN Physical Bit Transmission

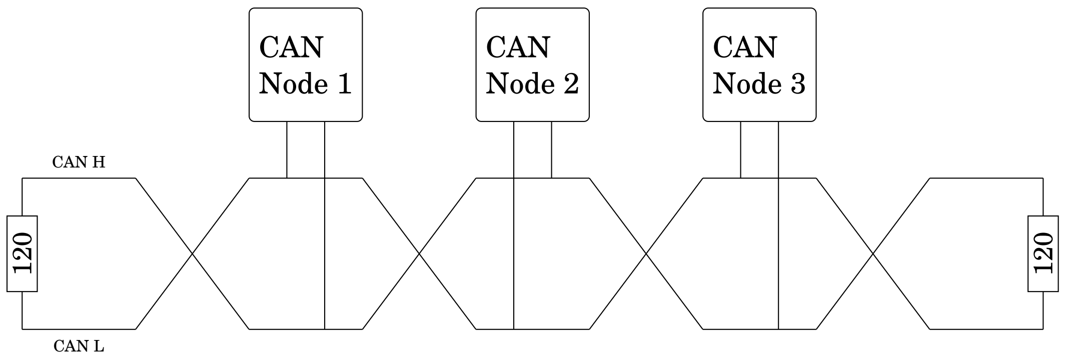


Figure 2.2: CAN Wiring

A CAN-transceiver, like shown in Figure 2.3 connects the controller to the bus. This transceiver converts the logic-level signals from the controller (CAN RX and CAN TX) to the bus levels and vice versa. The dominant timeout protects the bus from a persistent dominant state in case of a faulty CAN controller.

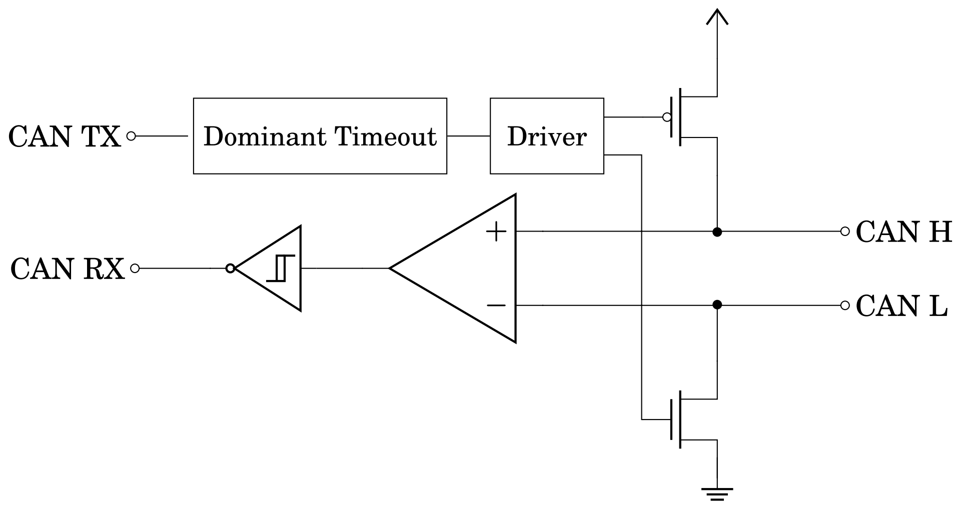


Figure 2.3: Simplified CAN Transceiver

The bitrate limit depends on the length of the bus [22]. However, the highest bitrate is limited to 1 Mbit per second for Classical CAN [5] and 8 Mbit for the data field of CAN Flexible Data-Rate (CAN FD)[6]. Table 2.1 shows some example length and bitrate combinations for Classical CAN. For CAN FD, the bitrate is only increased during the data field phase and could be eight times higher than the values in the table.

Table 2.1: Maximum bus speed

Length	Max. Speed
[m]	[Kbps]
40	1000
100	500
200	250
500	100
1000	50

2.1.2 CAN Frames

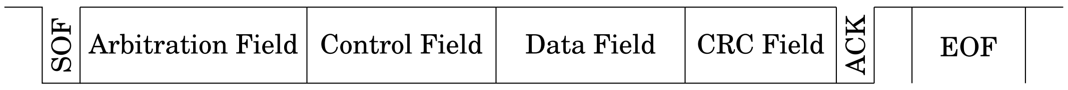


Figure 2.4: CAN Frame Format

The Medium Access Control (MAC) specification defines the CAN frame format, as shown in Figure 2.4. CAN uses so-called identifiers to identify the frames. Identifiers do not address nodes but identify the data that is being sent. There are four basic frame formats. The ”CAN Base Frame Format” with an 11-bit identifier; the ”CAN Extended Frame Format” with a 29-bit identifier and their FD variants. Frames start with the ”Start of Frame” (SOF) bit. This bit is always dominant and signalizes the start of the frame and synchronizes the nodes. The Arbitration Field includes the Base identifier and the ”Remote Transmission Request” (RTR) bit in case of a basic frame. For the extended frame format, the Identifier Extension (IDE) bit signalizes the identifier extension. In this case, the IDE-bit and the remaining 18 bits of the identifier are also part of the Arbitration Field. Nodes can use the RTR-bit to signal other nodes a request for transmission. For example, trigger a sensor read.

Figure 2.5 and Figure 2.6 shows examples for a Basic and Extended Frames. CAN FD frames are outlined in more detail in the Bosch CAN FD specification [6].

In the Arbitration Field, collisions of multiple sending nodes are allowed.

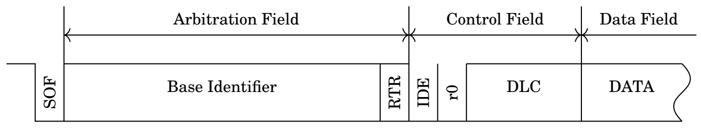


Figure 2.5: CAN Base Frame

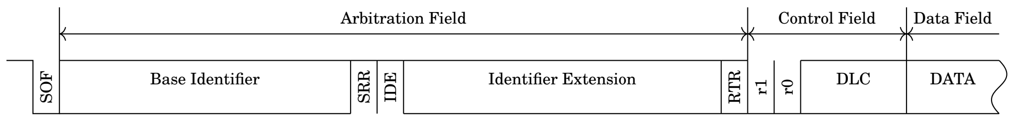


Figure 2.6: CAN Extended Frame

The fact that a dominant bit always overrides a recessive bit resolves collisions in a way that the frame with the lowest identifier always wins. Senders that want to write a recessive bit but get overridden by a dominant bit must abort their transmission silently. Aborted frames can be retransmitted when the bus is in the idle state again.

The Control Field includes the reserved bit r0, the IDE, and the Data Length Code (DLC) for Basic frames and reserved bit r1, r0, and the DLC for Extended frames. DLC indicates how many bytes are transmitted during the data phase. Table 2.2 lists the codes and number of bytes. The maximum number of bytes for Classical CAN frames is eight, and the maximum number of bytes for CAN FD is 64.

Table 2.2: Data Length Codes

DLC	Bytes
0-8	0-8
9	12
10	16
11	20
12	24
13	32
14	48
15	64

The data field can be either empty or as many bytes as indicated by the DLC.

After the data field, the CRC field follows. The length of the CRC field depends on the length of the data field. A 15-bit CRC is used for all CAN frames up to eight data bytes. For data field length up to 16 bytes, a 17-bit CRC is used. A 21-bit CRC is used for a data field length of more than 16 bytes. The CRC is calculated over the whole frame, from SOF to the end of the data field.

All nodes that received the frame correctly acknowledge their reception by putting a dominant bit into the ACK field. The Field after the ACK field is the ACK-Delimiter and is always recessive.

EOF is the ”End Of Frame”. It is always seven recessive bits. The Inter Frame Space is at least three recessive bits called Intermission. Any node can override this Intermission with dominant bits to signal an Overload condition.

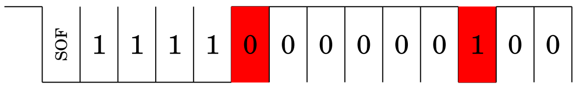


Figure 2.7: Bit Stuffing Example for 0x780 Identifier

The bit-stuffing mechanism provides reliable synchronization during the transmission of the frame. Five consecutive bits of the same value force a so-called stuffing bit. The stuffing bit is a bit with the inverted value of the bits before and must be ignored by the receivers. Stuffing bits change the length of the frame. An example is shown in Figure 2.7. In the example stuffing bits, highlighted in red, extends the identifier field to 13 bits. Additionally, it shows that SOF must be taken into account for stuffing.

Intentional violations of the stuffing rule are called Error Frames. Overriding six consecutive bits with a dominant bit is an Error Active Frame. Six recessive bits are called Error passive frame. This rule allows signaling error conditions during all phases, including the EOF, in case of a CRC mismatch.

2.2 ISO-TP Transport Protocol

ISO-TP is a short name for the transport protocol specified in ISO15765 [16]. It specifies a transport protocol and network layer service operating in Controller Area Networks. It was initially designed for road vehicle diagnostic services but it is not limited to them. ISO-TP overcomes the limited frame size of CAN frames and allows us to send packets up to 4095 bytes and in the latest version extended to 4 gigabytes. For this, the packets are segmented, transmitted in CAN frames, and reassembled on the receiver. Additionally, a flow-control mechanism is defined to steer the timing of the frames.

The header data is called Process Control Information (PCI). The first nibble is the PCI-Type. Following four PCI types are defined:

- Single Frame
- First Frame
- Consecutive Frame
- Flow Control Frame

The rest of the PCI depends on the PCI-Type. After the PCI, the rest of the frame is filled with payload data. Single frames are used when the payload data plus one byte for the PCI fits into a single CAN frame. For Classical CAN, it is a payload length of seven bytes. For CAN FD, it can be up to 63 bytes. The other three PCI types are used for payload lengths larger than a single frame can carry.

A segmented packet starts with a ”First Frame” (FF) from the sender to the receiver. The First Frame contains information about the total payload data length. The receiver sends back a ”Flow Control” frame (FC). This FC frame can either signal ”Continue To Send”, ”Wait”, or that the packet would ”Overflow” the receiver. Furthermore, the FC frame contains a Block Size (BS) and Minimal Separation Time (ST_{min}). When the FC frame is of type CTS, the sender continues with sending ”Consecutive Frames” (CF). When BS is zero, the sender sends as many CF frames as needed to transfer the payload data. Otherwise, the sender has to stop after BS CF frames and wait for another FC frame. ST_{min} defines a minimum separation time between two frames. A ST_{min} of zero is allowed. If the receiver answers with an FC-Frame with the Flow-State OVFLW, the sender has to abort the transmission. In case the sender does not receive an FC-Frame within one second, it aborts the transmission. The Flow-State WAIT causes a reset of the sender timeout. Receivers cancel the reception of the packet when no CF frame is received within one second.

Figure 2.8 shows an example sequence with a BS of three.

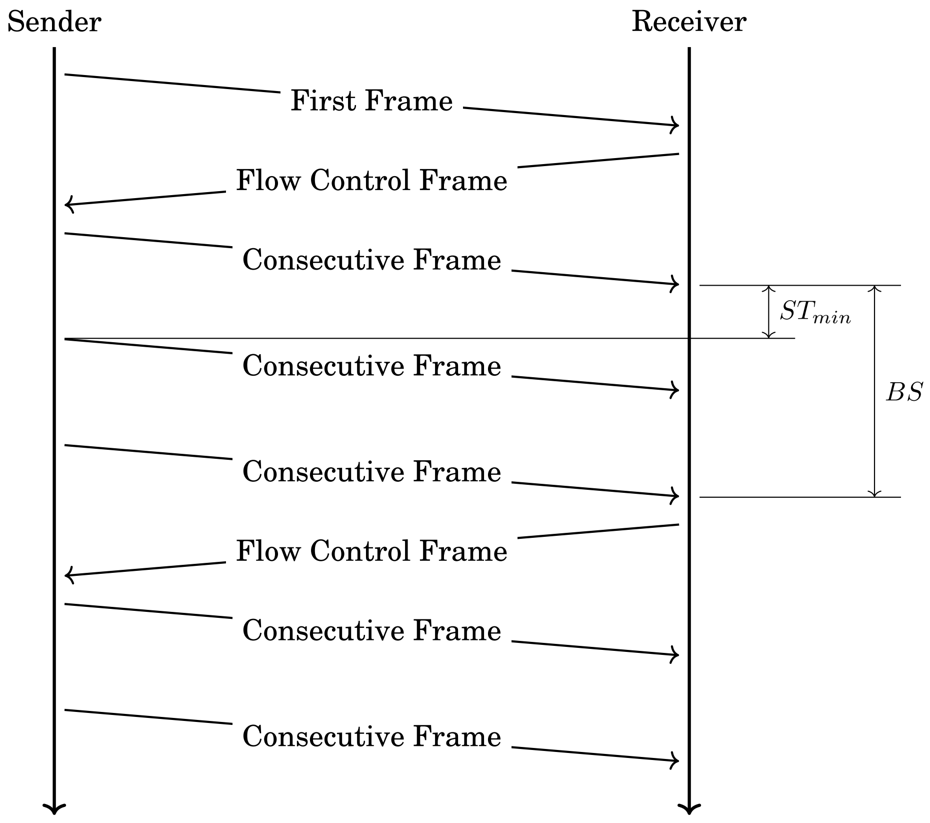


Figure 2.8: Example Sequence of a Segmented Packet with a BS of three

Table 2.3 shows examples of classical CAN frames with all PCI types. Elements labeled with data 0 to data 6 are payload data.

Table 2.3: ISO-TP Protocol Control Information

Byte:	0		1	2	3	4	5	6	7
	7..4	3..0							
SF	0	SF_DL	data 0	data 1	data 2	data 3	data 4	data5	data 6
		0	SF_DL	data 0	data 1	data 2	data 3	data 4	data 5
FF	1	FF_DL		data 0	data 1	data 2	data 3	data 4	data 5
		0		FF_DL				data 0	data 1
CF	2	SN	data 0	data 1	data 2	data 3	data 4	data 5	data 6
FC	3	FS	BS	STmin					

Table 2.4: Flow-States

Type	Number
CTS (Continue To Send)	0
WAIT (Wait)	1
OVFLW (Overflow)	2

SF is the Single Frame and has the PCI-type-number 0. For Single Frames with maximum payload data size of seven bytes, the Single Frame data length (SF_DL) is encoded in bit zero to three of the first PCI byte. CAN FD frames can have a frame data length up to 64 bytes in a single frame. Single frames with more than seven bytes of data encode the data length in the second PCI byte.

FF is the First Frame and has the PCI-type-number 1. The data length (FF_DL) of the packet is encoded in byte zero and byte one of the PCI. Byte one is the lower octet, and bit zero to bit 3 of byte 0 contains the upper nibble of the 12-bit data length. For a data length bigger than 4095 bytes, bit zero to three of byte zero and byte one is set to zero. The data length is then encoded in byte two to five.

CFs are Consecutive Frames, containing payload data and a sequence number (SN). The PCI-type-number is 2, and the sequence number is located at the lower nibble of byte zero. The sequence number is a counter that is set to one for the first CF and incremented by one for every frame and wraps around at 15. When the sequence number wraps around, it starts with zero again. The sequence number is used to detect out of order or lost frames.

FCs are Frame Control Frames and have the PCI-type-number 3. The lower 4-bit nibble of the first byte contains Flow State. The numbers are shown in Table 2.4. The second byte, Block-Size (BS), defines how many CF frames the sender is allowed to transmit until he has to wait for an FC-Frame again.

2.3 IPv6

IP (Internet Protocol) covers the Internet layer of the Internet Protocol Suite (Table 2.5) [4]. Internet Protocol version six (IPv6) was first introduced in RFC2460 [11] as an RFC Draft-Standard in December 1998 and got finally standardized with RFC8200 [7] in July 2017. It solves some problems from its predecessor, the Internet Protocol version four (IPv4) [14].

The most significant improvements are:

- Extended address range (from 32 bits to 128-bits)
- Stateless Address Autoconfiguration
- Simplification of the header
- Use of Neighbor Discovery protocol
- Next header instead of header options

RFC8200 defines some important terminologies: A ”node” is a device with an IPv6 stack. ”Link” refers to the lowest layer defined in Table 2.5, and ”interface” is the node’s attachment to a link.

Table 2.5: Internet Protocol Suite

Layer	Example
Application	HTTP, HTTPS, CoAP, MQTT
Transport	TCP, UDP
Internet	IPv4, IPv6, ICMP
Link	Ethernet, IEEE 802.15.4, CAN

Table 2.6: IPv6 Address Types

Type	Prefix
Unspecified Address	::/128 (all zero)
Loopback	::1/128
Multicast	FF00::/8
Link-Local Unicast	FE80::/10
Global Unicast	everything else

2.3.1 IPv6 Addresses

IPv6 addresses are written in a hexadecimal representation of 16-bit blocks, separated by colons. Leading zeros can be omitted, and all zero blocks can be written as two colons (::). The all-zero blocks can span more than a single 16-bit block, but can only be used once in a representation. RFC5952 [18] describes the recommended text representation of IPv6 addresses.

For example, the address fe80:0000:0000:0000:0000:00ff:fe00:1234 can be written as fe80::ff:fe00:1234

An IPv6 address prefix can be written as *IPv6-address*/*prefix-length*. For example fe80::00ff:fe00:1234/64 is interpreted as fe80:000:0000:0000 address prefix.

RFC4291 [12] defines five types of addresses, identified by their higher-order bits. The address types are shown in Table 2.6.

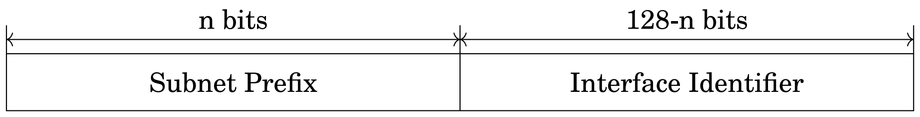


Figure 2.9: Subnet Prefix of an IPv6 address

Unicast IPv6 addresses can generally be treated as if they have no internal structure. Nevertheless, more sophisticated nodes may be aware of subnets, which logically group addresses. Figure 2.9 shows how the address is split into the subnet prefix and an Interface Identifier (IID). Routers, for example, can use subnets to create hierarchical boundaries. The IID needs to be unique within the subnet and identifies an interface on a link (lowest layer on Table 2.5). All Unicast addresses not starting with the binary value 000 must have an IID according to the Modified EUI-64 format with a length of 64 bits. Addresses of this format can be derived from the link-address as described in RFC4291 [12] Appendix A. If the address is derived, for example, from an Ethernet MAC address, it has a universal scope and is globally unique. IIDs derived from link addresses that are not globally unique must be unique in the local scope.

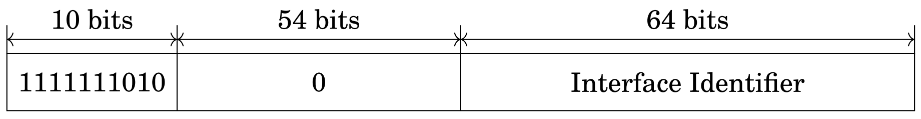


Figure 2.10: Link Local Unicast address

An interface can have several addresses, but must at least have one Link-Local Unicast address. The Link-Local Unicast address is formed as shown in Figure 2.10.

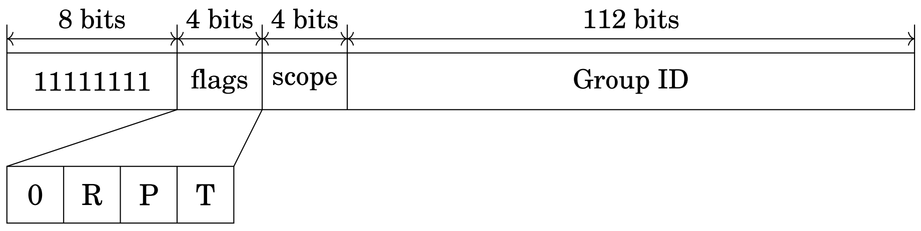


Figure 2.11: Multicast address

The multicast address format depicted in Figure 2.11. The highest byte is the multicast prefix 0xff, followed by four bits flags, and four bits scope. For the flags, only the T flag is of interest in this work. The T-flag signals an unassigned address when it's set to one. If the T-flag is zero, the address is a ”well-known” address. The scopes are listed in Table 2.7.

Table 2.7: Multicast Scopes

Number	Scope
0	reserved
1	Interface-Local
2	Link-Local
3	reserved
4	Admin-Local
5	Site-Local
6-7	unassigned
8	Organization-Local
9-D	unassigned
E	Global
F	reserved

Some predefined multicast addresses are, for example:

- ff01::1 Interface-Local All-Nodes multicast address
- ff02::1 Link-Local All-Nodes multicast address
- ff02::2 Link-Local All-Routers multicast address
- ff02::2 Site-Local All-Routers multicast address
- ff02:0:0:0:0:1:ffXX:XXXX Solicited-Node multicast address

The Solicited-Node multicast address is formed by the prefix ff02:0:0:0:0:1:ff::/104 concatenated with the low-order 24 bits of the unicast or anycast address.

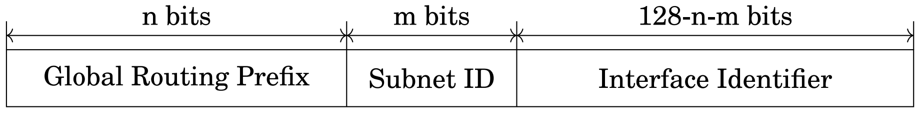


Figure 2.12: Global Unicast address

A Global Unicast Address, as shown in Figure 2.12 consists of a global routing prefix, a subnet ID, and the IID. Global unicast addresses not starting with binary 000 always have an IID length of 64 bytes, same as the Link-Local Unicast address. The global routing prefix is a value assigned to a site and distributed by the routers on that site. The Subnet ID is an identifier of a link within the site. The global unicast address is used to address single nodes that are not Link-Local, for example, a node on the internet.

A node must at least listen to the following addresses:

- The Link-Local addresses assigned to the interfaces
- The loopback address
- The All-Nodes multicast address
- The Solicited-Nodes multicast address for each anycast and multicast address
- All multicast addresses for multicast groups the node has joined

2.3.2 IPv6 Header

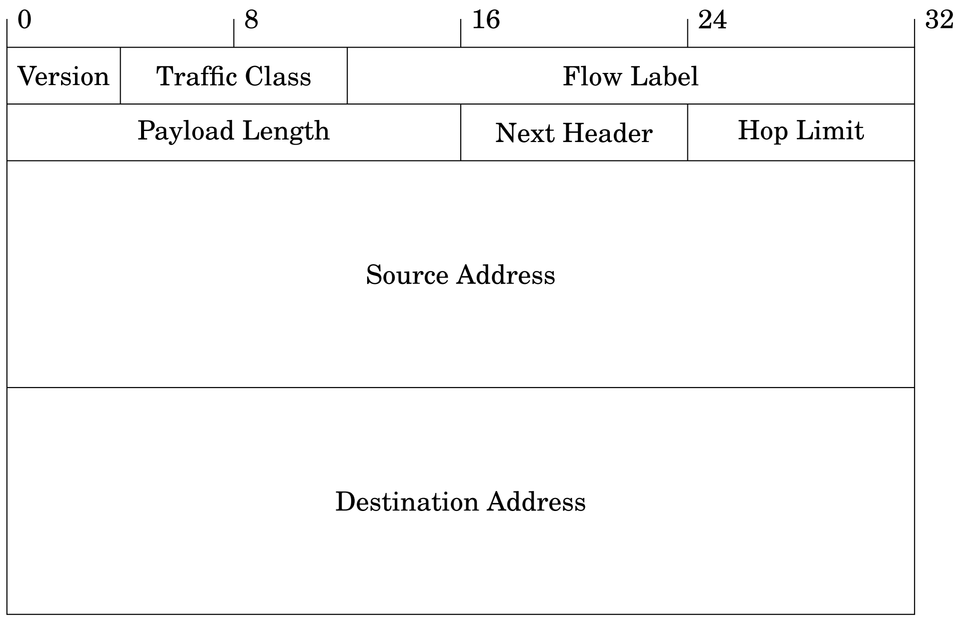


Figure 2.13: IPv6 Header

The IPv6 header, as shown in Figure 2.13, has a fixed size of 40 bytes. The fixed length makes it easier to process IPv6 headers and allows simple compression schemes like IPHC [24].

The 4-bit Version field is always 0x6 for IPv6. Traffic-Class has a length of eight bits where the six most significant bits correspond to the Differentiated Service (DS) [3] and the remaining two bits are used for Explicit Congestion Notification (ECN) [8]. The 16-bit Payload Length field is an unsigned integer, defining the remaining length of the packet. This number excludes the IP header but includes all possible next headers. The next header field identifies the header type of the following extension- or protocol-header, if any, or ”No Next Header” (59) if there aren’t any.

The concept of Next Headers instead of options makes IPv6 very flexible. Any number of options headers may follow the IPv6 header in a chained manner. Therefor, every option header includes another next header field, forming a chain until the next header is an upper-layer protocol or the ”No Next Header” option.

The Hop Limit is a counter, initialized by the node that issued the packet, with the desired limit of how many times this packet may be forwarded. Every hop that forwards, a router, for example, decrements the hop limit by one. When the limit reaches zero, the packet is discarded, and the node sends a message back to the source to inform about the discarding.

The Source Address is the IP address of the interface, and the Destination Address is the IP address of the desired receiver. The Source Address and Destination Address both have a length of 128 bits.

IPv6 supports fragmentation, where a packet is cut into smaller fragments at the sender, transported, and reassembled at the destination. For fragmentation, the Fragment Header is used. Packet fragmentation is only allowed at the sender. Routers are not allowed to fragment packets. IPv6 requires a minimal Maximum Transfer Unit (MTU) of 1280 bytes, which is the minimum link MTU that must be supported by every node on the internet. Links that are not capable of transferring packets of at last 1280 bytes must apply fragmentation and reassembly at the Data-Link Layer.

Table 2.8: ICMPv6 Types defined by RFC4443

Type	Name	Description
1	Destination Unreachable	Packet cannot be delivered to its destination
2	Packet Too Big	Router cannot forward the packet because it exceeds a link-MTU
3	Time Exceeded	Router received a packet with a hop-limit of zero or decremented it to zero
4	Parameter Problem	Node detected a problem on a header field that cannot be resolved
128	Echo Request	A request to reply back to the sender
129	Echo Reply	The reply to the Echo request

2.3.3 ICMPv6

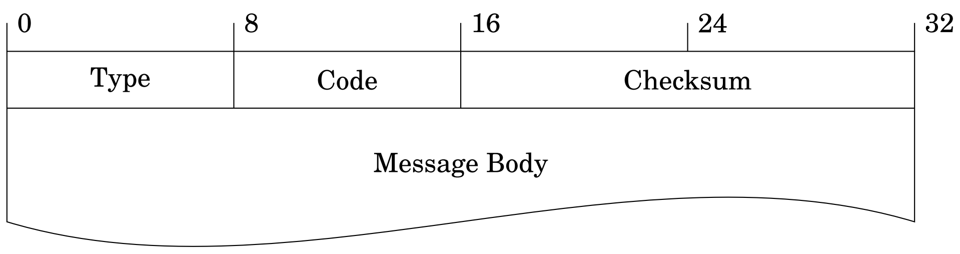


Figure 2.14: Internet Control Message Protocol Format

ICMPv6 is a Transport-Layer protocol defined in [10]. Figure 2.14 depicts the ICMPv6 header format. Table 2.8 shows ICMPv6 Types defined directly by the ICMPv6 RFC. Other types are defined in the respective RFCs defining ICMPv6 messages.

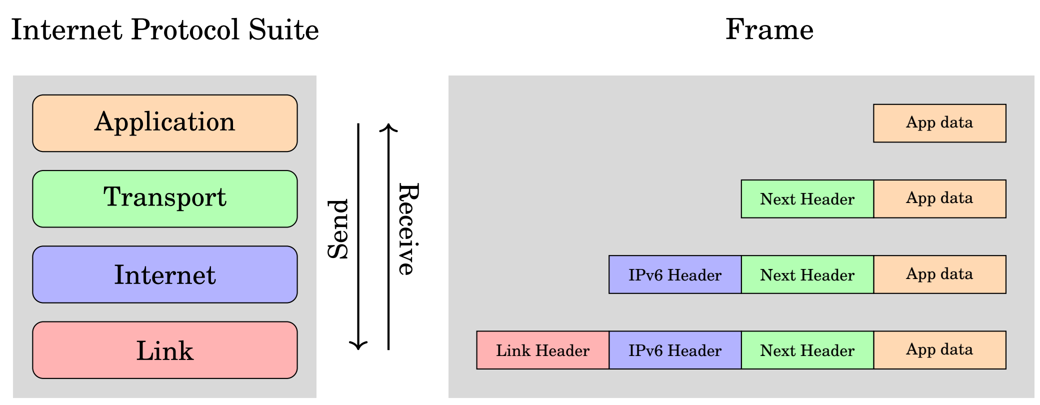


Figure 2.15: IPv6 Header Encapsulation

2.3.4 Neighbor Discovery Protocol

Figure 2.15 shows how the data and headers of the upper layers get encapsulated in an IPv6 datagram that gets finally encapsulated in a frame. IPv6 works on 128-bit addresses, but nodes use different addresses on their Link-Layer to transmit the frames. This so-called Link-Layer address could either be the address of the receiver directly on the link or the Link-Layer address of the router in case the receiver is not in the same Link-Local network. The Link-Layer address of the next hop or the destination, in case of a Link-Local transfer, needs to be discovered first. IPv6 uses the Link-Layer Neighbor Discovery protocol (NDP) [21] for this task. NDP packets are encapsulated in ICMPv6 (Internet Control Message Protocol) messages. The nodes use a so-called Neighbor Cache to learn the relevant addresses of its neighborhood. Addresses that have been resolved once stay in the cache for a predefined time. For this time, the address does not need to be resolved but can be taken from the cache.

NDP defines two message formats for finding direct neighbors.

- Neighbor Solicitation Message Format
- Neighbor Advertisement Message Format

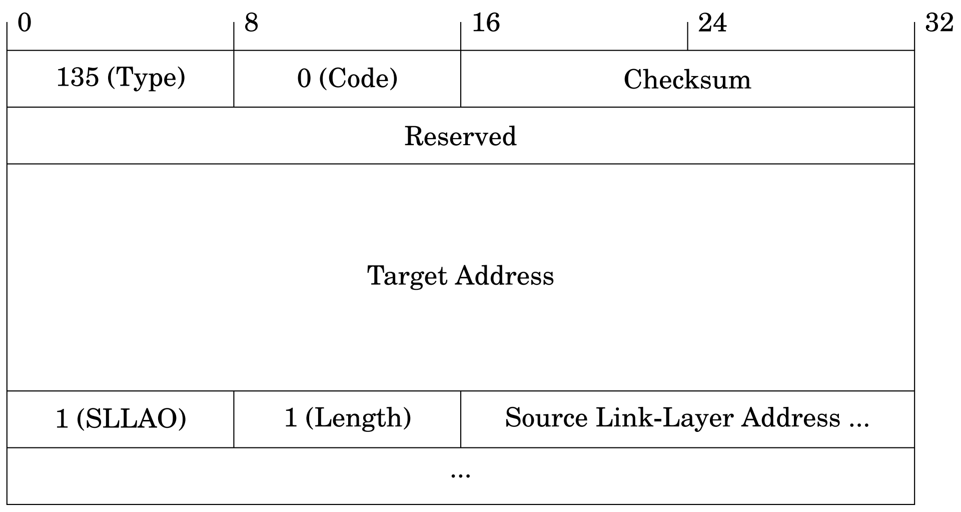


Figure 2.16: Neighbor Solicitation Message Example

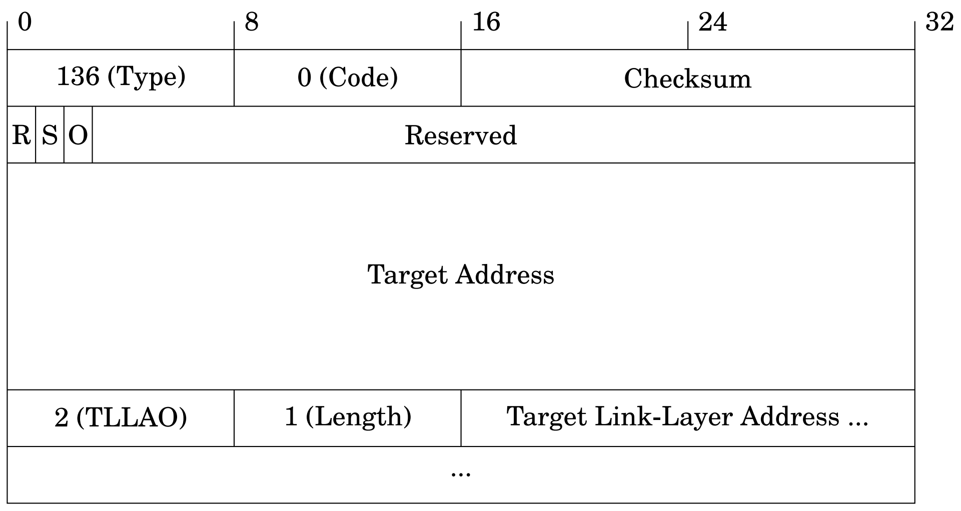


Figure 2.17: Neighbor Advertisement Message Example

Neighbor Advertisement messages propagate a node's Link-Layer address to other nodes. They are broadcasted when the node joins a network or can be requested by a Neighbor Solicitation message (NS). NAs not requested by an NS are called unsolicited advertisements and typically sent to the all-node multicast address. The node can send an NA message to the solicited-node multicast address and wait for an NA from the targeted node to resolve an address. The node can send an NS message to the targeted nodes unicast address to check if the node is still available.

Figure 2.16 shows an example of a Neighbor Solicitation message. The ICMPv6 Type of an NS is 135, and the code is always zero for an NS message. The Target Address field contains the IPv6 address of the node, where the NA should be requested. The Target Address is followed by the option field. In case the source of the IP header is not the unspecified address (::), the option field must include the Source Link-Layer Address option (SLLAO). The length field of a Link-Layer Address Option (LLAO) describes the total length of the option field in eight-bytes units. For example, if the Link-Layer address has a length of six bytes or smaller, the length field is one.

The answer to an NS message is the Neighbor Advertisement message (NA). Figure 2.17 depicts an example of such an NA message. The ICMPv6 type field is always 136, and the code is always zero for an NA message. The R-bit implies that the node, sending the NA, is a router. A set S-bit means that the NA message is sent as a response to an NS message. The O-bit indicates that this NA message should override a possibly cached Link-Layer address from a previously sent NA. The Target Address can either be the address of the solicitation node or the all-nodes multicast address in case of an unsolicited NA or an unspecified source address.

NDP defines two message formats for finding routers on the link.

- Router Solicitation Message Format
- Router Advertisement Message Format

Router advertisement messages (RA) propagate information about routers on the link. Unsolicited router advertisements are sent out to the all-nodes multicast address by the routers periodically. A node can also request a router advertisement by sending out a Router Solicitation (RS) message to

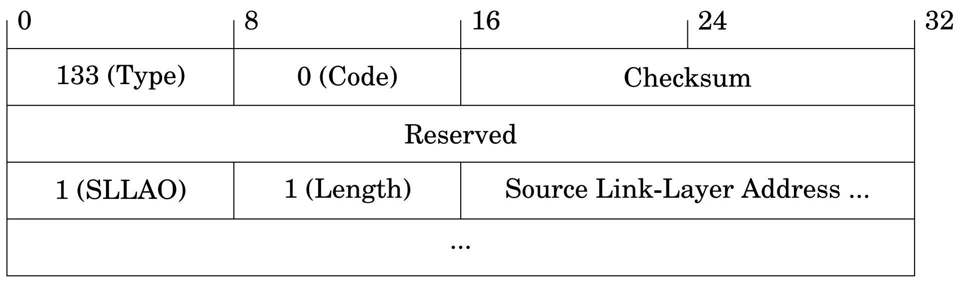


Figure 2.18: Router Solicitation Message Example

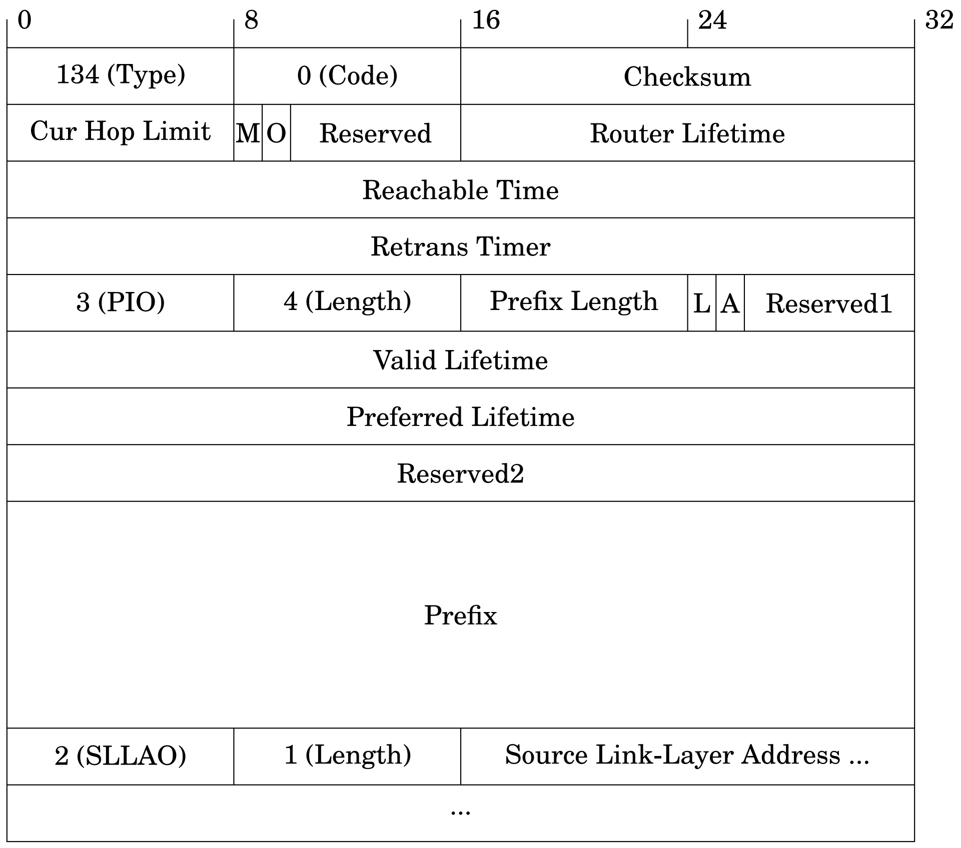


Figure 2.19: Router Advertisement Message Example

the all-routers multicast group. The router will then answer with an RA to the issuer's unicast address. The RS message is shown in Figure 2.18.

The RA message is shown in Figure 2.19. The ICMPv6 Type of an RA is 134, and the code is always zero for an RA message. The Current Hop Limit field tells the receiving node which hop-limit should be used for outgoing packets. Zero means unspecified. The M and O flags are DHCPv6 related and not relevant for this work. The Router Lifetime field is an unsigned 16-bit value in units of a second. It specifies the lifetime associated with the default router. The Reachable Time specifies a time interval in milliseconds for which a node can assume that an already discovered node is still reachable. The last timer field, the Retrans Timer, defines the time between the retransmission of a Neighbor Solicitation message in units of milliseconds. The RA example has two option fields included. The Prefix Info and the SLLAO. The SLLAO includes the source Link-Layer address like the RS message. The Prefix Info option (PIO) tells the receiving node information about the prefix used for address autoconfiguration. The length value indicates the number of bits used as the prefix. The prefix is always in the upper address bytes since the address in this option has the same length ass an entire address. The L-Flag indicates that the prefix is on-link, which means that it can be reached directly without any router involved. The A-Flag signals the prefix can be used for Stateless Address Autoconfiguration. Valid Lifetime and Preferred Lifetime signal the SLAAC for how long the address is valid and for how long it should be used. All-ones (0xffffffff) values signal infinite lifetime.

2.3.5 Stateless Address Autoconfiguration

IPv6 nodes use the Stateless Address Autoconfiguration (SLAAC), defined in RFC4862 [23], to configure the interface's addresses. The SLAAC mechanism generates Link-Local and global addresses. The Link-Local address is formed by combining the Link-Local prefix with the IID, and global addresses use the prefix learned from RA messages, combined with the IID. Figure 2.20 shows the routine to perform SLAAC. When an interface is initialized, it assigns itself a tentative Link-Local address, formed as described above. Then the node sends out an NS message with the Target Address of

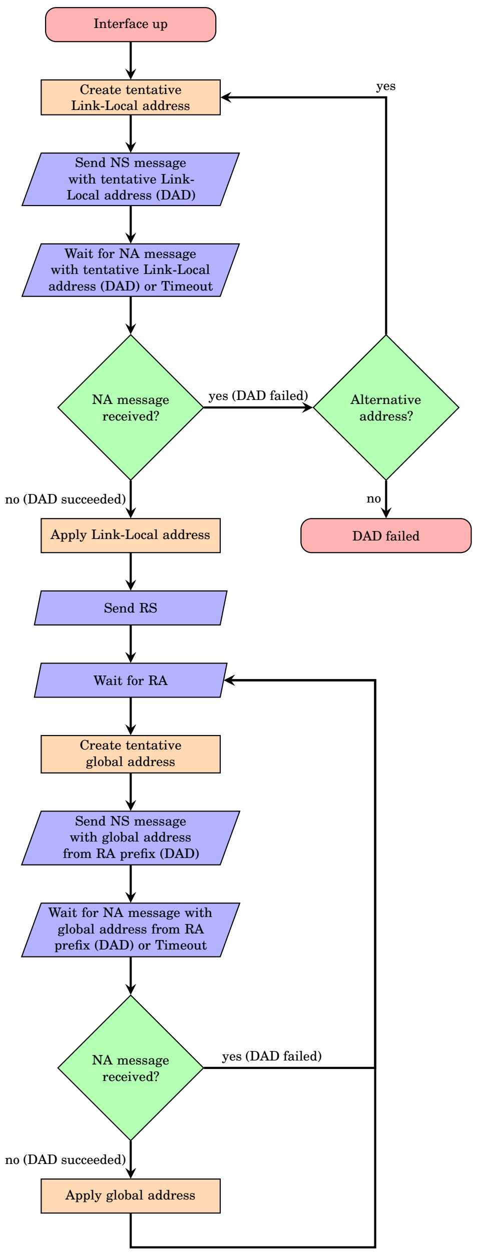

Interface up
Create tentative Link-Local address
yes
Send NS message with tentative Link-Local address (DAD)
Wait for NA message with tentative Link-Local address (DAD) or Timeout
NA message received?
yes (DAD failed)
Alternative address?
no (DAD succeeded)
no
Apply Link-Local address
DAD failed
Send RS
Wait for RA
Create tentative global address
Send NS message with global address from RA prefix (DAD)
Wait for NA message with global address from RA prefix (DAD) or Timeout
NA message received?
yes (DAD failed)
no (DAD succeeded)
Apply global address

Figure 2.20: Stateless Address Autoconfiguration

the tentatively chosen address. The source IPv6 address is the unspecified address (::) and the destination IPv6 address is the solicited multicast address. This process is called Duplicate Address Detection (DAD). If there is a node on the link, using the same address, it returns an NA message, indicating a duplication, and the node then discards the tentative address and depending on the configuration, generates a new address. After the DAD, the node has a Link-Local address assigned and can send an RS. All routers in the network answer the RS with an RA message where the node can get the prefix information from. With this prefix information, the node forms a global address and starts listening to that address.

2.4 6lo

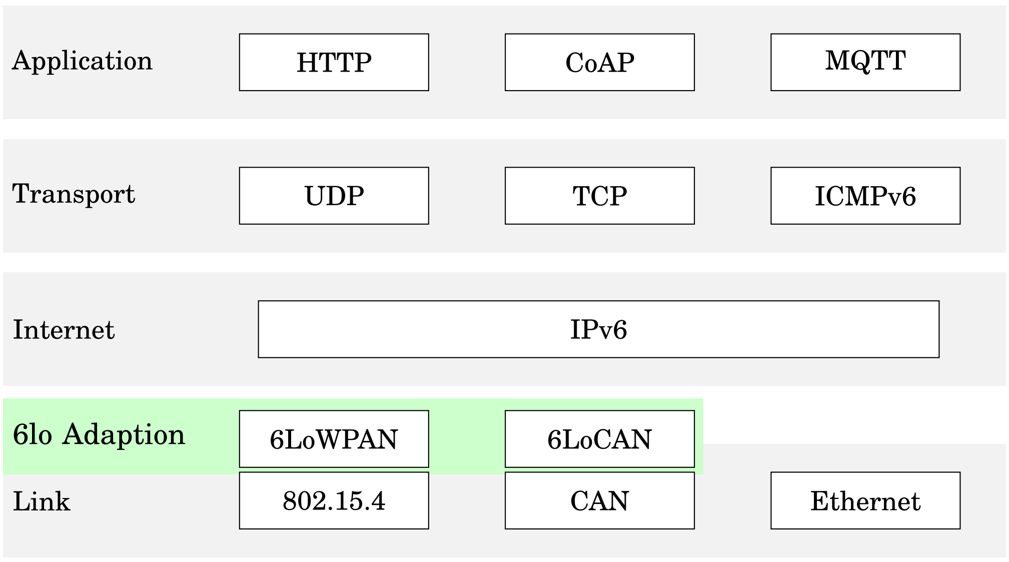


Figure 2.21: 6lo Adaption layer example

6lo is the name of an Internet Engineering Task Force (IETF) Working Group (WG). The 6lo WG focus on IPv6 connectivity over constrained nodes. The nodes are constrained in the sense of limited power, memory, and lack of some required Link-Layer services. The WG is working on IPv6-

over-foo adaption layers, using specifications from 6LoWPAN technologies like RFC4944 [19] and RFC6282 [24]. 6lo technologies usually have a 6lo adaption-layer, as shown in Figure 2.21.

2.4.1 IP Header Compression

For this document, the concept of the Adaption layer and the IP Header Compression (IPHC) is of importance. The IPHC reduces redundant information in the 40 bytes IPv6 header. The payload length, for example, is usually known from the Link-Layer packet size and is therefore always elided. The version field is also always elided because the compression is only defined for IPv6. Fields that can be recovered from Link-Layer information can be fully elided. Fields filled with default parameters can either be entirely elided or chosen on a set of default values. On a Link-Layer where the Interface Identifier (IID) is generated from the Link-Layer address, the IPv6 header with a Link-Local source and Link-Local destination address, the IPv6 addresses in the header are redundant and can be reconstructed from the Link-Layer headers. This saves up to 38 bytes of data. The compression can either be stateless or context-based. A context is a global routing prefix that is known by all nodes and has an index. This 4-bit index can then be used instead of transmitting the 64-bit prefix. The distribution of the contexts is out of scope and not defined by aby standard yet.

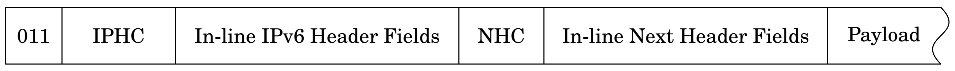


Figure 2.22: IPHC Frame Format

The IPHC dispatch is a bit-field at the beginning of a packet, as shown in Figure 2.22. The dispatch signals a IPHC compressed packet. IPv6 header fields that cannot be fully elided are carried in-line after the IPHC bit-field. The order of the in-lined data follows the order of the IPHC bit-field from left to right. If the Next Header is compressible, the Next Header Compression (NHC) field follows the In-Line Header Fields. Next-Header fields that cannot be fully elided, follow the NHC.

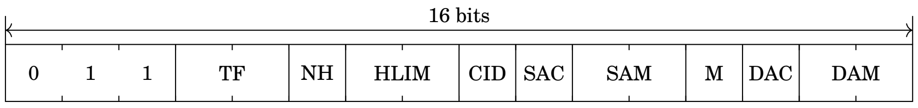


Figure 2.23: IPHC Format

The IPHC bit-field, including the three-bit dispatch (011), has a length of 16 bits, as shown in Figure 2.23.

TF (Traffic-Class, Flow Label):
The first two bits define the Traffic-Class and Flow Label compression (TF). The meaning of the values is stated in Table 2.9.

Table 2.9: Traffic-Class and Flow-Label Compression

Code	Description
00	ECN + DSCP + 4-bit Pad + Flow Label (4 bytes)
01	ECN + 2-bit Pad + Flow Label (3 bytes), DSCP is elided.
10	ECN + DSCP (1 byte), Flow Label is elided.
11	Traffic-Class and Flow-Label are elided.

NH (Next Header):
When this bit is set, the next header field is compressed, and the LOWPAN_NHC (Next header compression) is following the in-line field. Otherwise, the Next header byte is carried in-line.

HLIM (Hop Limit):
The HLIM field defines the compressed Hop Limit.

Table 2.10: Hop Limit Compression

Code	Description
00	No compression. Carried in-line
01	The hop limit is 1
10	The hop limit is 64
11	The hop limit is 255

CID (Context Identifier Extension):
If set, an 8-bit Context Identifier is carried in-line. The format is shown in Figure 2.24. If a context-based compression is used for source or destination-address, context zero is used.

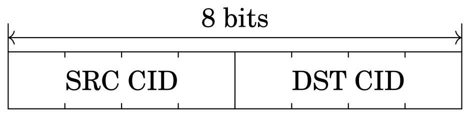


Figure 2.24: Context ID Format

SAC (Source Address Compression):
If set, the source address compression is context-based. If not, it is stateless.

SAM (Source Address Mode):
The source address compression mode is interpreted differently for context-based or stateless compression.

The enumeration for stateless source address compression is shown in Table 2.11.

Table 2.11: Stateless Source and Destination-Address Compression

Code	Description	In-lined bits
00	No compression. Address is carried in-line	128
01	The Prefix is fe80::/64. The IID is carried in-line	64
10	The Prefix is fe80::ff:fe00:XXXX/112. The last 16 bits are carried in-line	16
11	The Prefix is fe80::/64. The IID is reconstructed from the Link-Layer address.	0

The enumeration for context-based source address compression is shown in Table 2.12.

Table 2.12: Context-Based Source- and Destination-Address Compression

Code	Description	In-lined bits
00	The unspecified address ::	0
01	The Prefix is taken from the context. The IID is carried in-line.	64
10	The Prefix is taken from the context. Bits not covered by the context are filled with ::ff:fe00:XXXX. The last 16 bits are carried in-line	16
11	The Prefix is taken from the context. The IID is reconstructed from the Link-Layer address, if not covered by the context.	0

M (Multicast Compression):
If set, the destination address is a multicast address.

DAC (Destination Address Compression):
If set, the destination address compression is context-based. If not, it is stateless.

DAM (Destination Address Mode):
The interpretation of the destination address compression mode depends on the combination of multicast compression and destination address compression.

The enumeration for stateless non-multicast compression is shown in table Table 2.11 and is the same as the compression for the source address.

The enumeration for context-based destination address compression is the same as the compression for the source address, with the exception that case 00 is not allowed, and is shown in Table 2.12.

The enumeration for stateless multicast destination address compression is shown in

Another significant compression is the Next Header Compression, specifically the UDP header compression. The format is shown in Figure 2.25. The first five bits are the dispatch for UDP header compression.

Table 2.13: Stateless Multicast Destination-Address Compression

Code	Description	In-lined bits
00	No compression. Address is carried in-line	128
01	The address takes the form ffXX::XX:XXXX:XXXX	48
10	The address takes the form ffXX::XX:XXXX	32
11	The address takes the form ff02::XX	8

Bytes marked as XX are carried in-line.

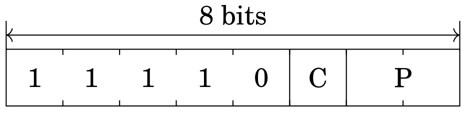


Figure 2.25: Next Header Compression for UDP Headers

C (Checksum): If C is set, the Checksum of the UDP datagram is elided or carried in-line otherwise.

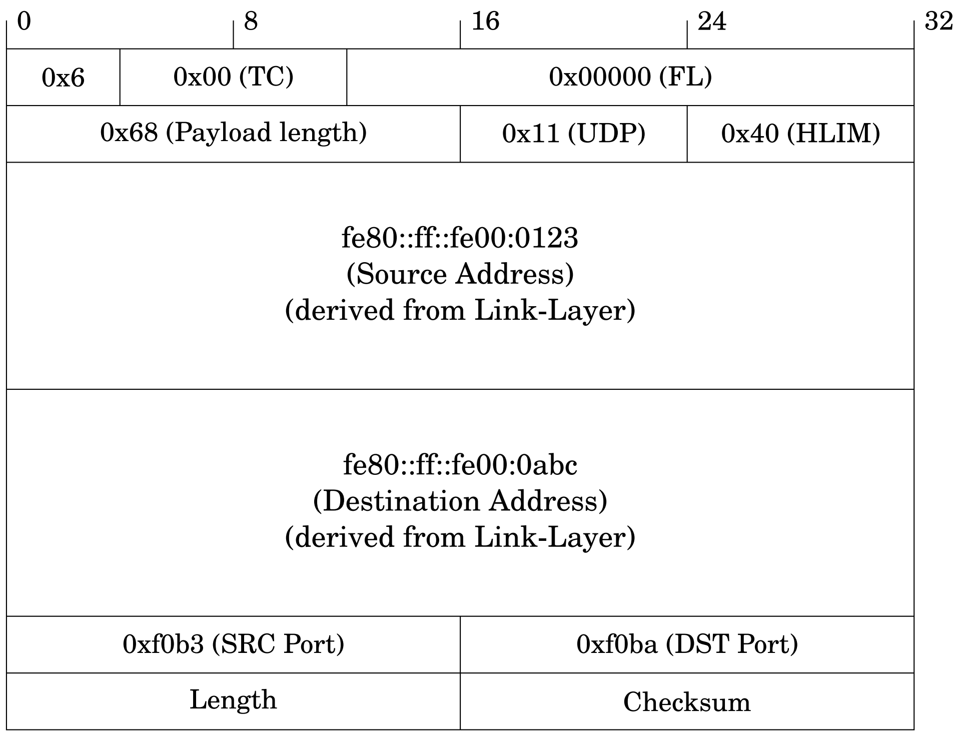
P (Ports): The enumeration of the port compression is shown in Table 2.14

Table 2.14: Stateless Multicast Destination Address Compression

Code	Description	In-lined bits
00	No compression. Both ports are carried in-line	32
01	Source-Port fully in-lined. Destination port takes the form 0xf0XX	24
10	Destination Port fully in-lined. Source port takes the form 0xf0XX	24
11	Source- and Destination Port take the form 0xf0bX	8

Bytes marked as XX are carried in-line.

Figure 2.26 depicts an optimal compression of a Link-Local IPv6 packet with UDP. The Traffic-Class (TC) and Flow-Label (FL) field can be fully elided because they are zero. All-zero TC and FL are a realistic scenario for Link-Local traffic and even for the most routed traffic. The UDP Next-Header-Field can be elided because we can use Next-Header-Compression



48 bytes from the original header result in following 4 bytes:

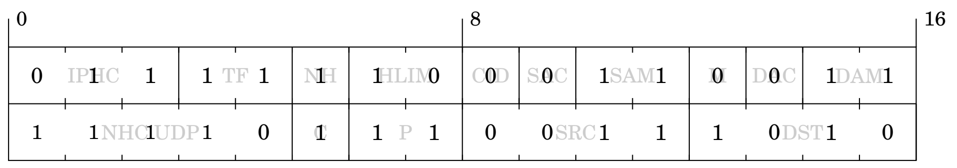


Figure 2.26: Link-Local IPv6 an UDP header to IPHC example

for UDP. A Hop-Limit of 64 can be compressed with no in-line data added. The IPv6 Source Address and destination address can be compressed to zero in-line data. The Link-Local prefix can be fully elided, and the IID can be reconstructed from the Link-Layer address. The UDP header is compressed by using the UDP Next Header Compression. The source and destination ports are chosen in a way that they can be efficiently compressed to a single byte. This efficient compression works for addresses of form 0xf0bX, where the last nibble can be chosen. The checksum is elided in this example. It can be elided if the link-layer already has a mechanism for checking the correctness of the data.

Efficient compression heavily depends on the IPv6 address and ports used. To have an efficient compression, the IDD should be derived from the Link-Layer address, and the prefix should either be link-local, or a context should be distributed among all nodes. RFC6282 does not define how to distribute the IPHC context; neither does this work do. Applications that are aware of the fact that IPHC is used, should use ports from 0xf0b0 to 0xf0bf to allow efficient UDP Next Header Compression.

2.5 Zephyr

Zephyr is a Real-Time Operating System (RTOS) launched by Intel and now hosted by the Linux Foundation as the Zephyrproject. The kernel is derived from WindRiver's Rocket kernel, and the Zephyr project source code is publicly available on GitHub under the Appache 2 license. The development is community and industry-driven with vendor-neutral governance. Major System on Chip (SoC) vendors are members of the project and provide funding. Zephyr is designed for small scale connected devices, but it can also run on an x68 application processor. Is supports various architectures like ARM, x64, ARC, RISC-V, and Xtensa. For each architecture, a set of SoCs and development boards are supported.

2.6 Zephyr Network Stack

Zephyr has a native IPv4 and IPv6 network stack built-in. The stack is authored and maintained by the Zephyr community. It supports TCP and UDP transport and all ICMP packets required by the IP standards. The interface provided to the user is a POSIX Sockets API. Currently the following Link-Layers are supported:

- Ethernet
- 6LoWPAN (IEEE 802.15.4)
- IPSP (Bluetooth Low Energy)
- 6LoCAN (Controller Area Network), as an outcome of this work

Figure 2.27 depicts an overview of the Zephyr stack. On the bottom, there are the implementations of the low-level network device drivers. They are responsible for handling the interrupts, sending, and receiving raw data.

The next layer is the Link-Layer Technologies Layer. This layer takes care of the Link-Layer headers, and in the case of 6lo Technologies, performs fragmentation, reassembly, IPv6 header-compression, and uncompression.

The Network Core Layer is kind of a dispatcher for packets. It accepts raw packets from the network and put them into the networking working queues, depending on the packet priority. This layer is the boundary between the driver context, that could possibly be an interrupt, and networking context. The networking context consists of multiple work-queue threads for receiving and sending. The work-queue threads have different priorities, to support traffic classification by priority. The number of threads can be chosen by a configuration option.

The Network Interface Abstraction Layer defines a common API for Link-Layer implementations, handles the IP addresses of the interface and provides a generic way to read the Link-Layer addresses.

The Networking Protocols Layer parses the IP headers, checks the addresses, and decides if the packet is addressed to this node or should be discarded. It also parses the next header and invokes the corresponding transport protocol.

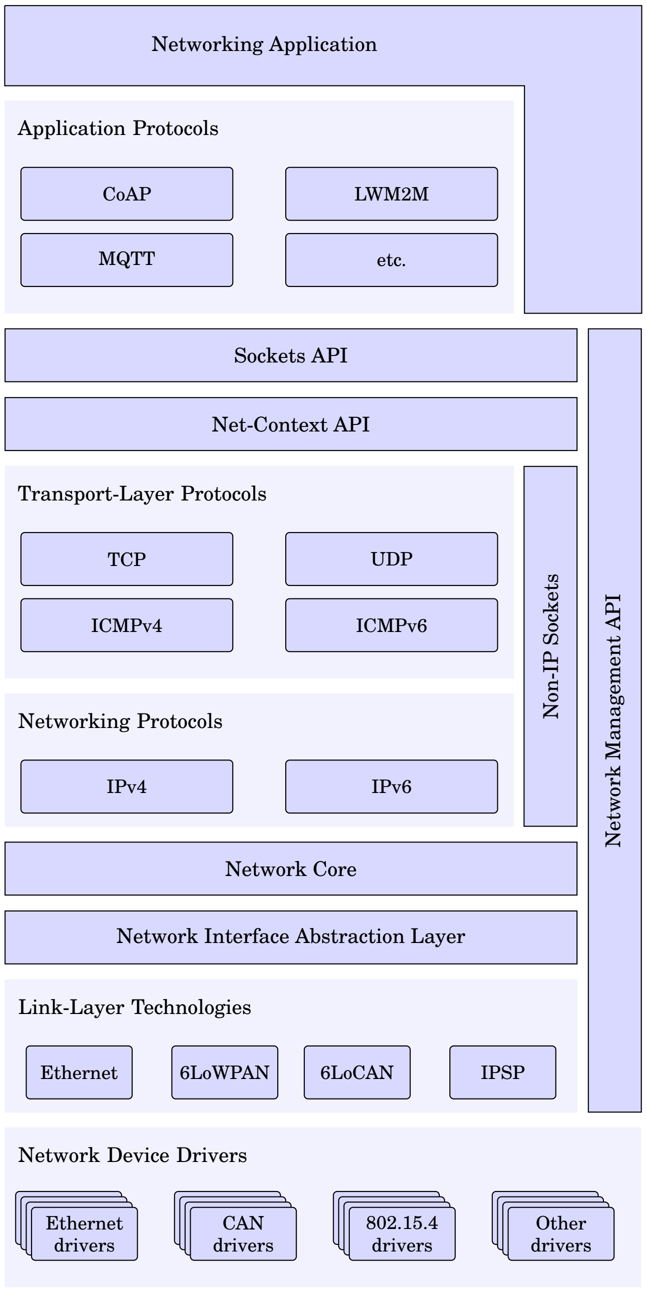


Figure 2.27: Zephyr Network Stack 6LoCAN RX example

The Transport-Layer Protocols Layer parses and validates the transport layer protocol header. In the case of ICMP, the packets are handled by this layer, while TCP and UDP packets are handed over to the Net-Context API by the IP layer.

The Net-Context API layer checks for registered handlers and if a handler is found, forwards it to them. The Sockets API registers the handlers, that takes the packet from the Net-Context and forwards it to the user application. The Sockets API follows the POSIX [13] BSD sockets API standard.

3 6LoCAN design

The goal of 6LoCAN is to support IPv6 traffic over the CAN-bus with as little overhead as possible. The payload-data size for classical CAN is eight bytes and for CAN-FD it is 64 bytes. To satisfy the minimum MTU requirement of 1280 bytes, an efficient fragmentation and reassembly mechanism is required.

Sending IPv6 headers over Classical CAN would at least require six CAN frames, assuming the fragmentation and reassembly header-only takes one byte. Six frames for only sending the header is not efficient, and therefore, a header compression algorithm is applied. In the current design, IPHC from 6LoWPAN is used. In an optimal scenario, the IPv6 and UDP header can be compressed to a size as small as four bytes.

For fragmentation and reassembly, we are using the well known ISO-TP protocol. ISO-TP can transfer packets with a size of up to 4095 bytes, which is sufficient for the minimal MTU requirements of IPv6. It also provides a flow-control mechanism for unicast transfers. ISO-TP does not support multicast transfers, and therefor 6LoCAN uses a slightly modified version that has some simplifications and allows a multicast transfer. The fragmentation headers designed for 6LoWPAN have a size of four bytes for the first fragment and five bytes for consecutive frames [19]. For classical CAN, this is more then the half of the frame payload, which is not efficient enough, and therefore the decision was made not to use it.

CAN uses identifiers to identify frames instead of addresses to address nodes. However, or IPv6 traffic, we need a way to address dedicated nodes. For this purpose, we introduced an addressing schema that translates node addresses to identifiers. This schema uses the 29-bit addresses only, and the 11-bit identifiers can still be used for other traffic than 6LoCAN. The node-addresses defined by the addressing schema have a length of 14 bits and can either be statically assigned or randomly chosen during the initialization of the interface. Node-addresses need to be unique on the bus. To verify the uniqueness of the node-address, we introduced a Link-Layer duplicate address detection.

This work also defines a translator from the 6LoCAN network to an Ethernet-based network. We named this translator mechanism a 6LoCAN border translator. The translator makes it possible to connect 6LoCAN nodes with Ethernet nodes on the same Link-Local domain. The translator can then be connected to another node, an Ethernet-Switch, Router, or whatever device using Ethernet. The translator is stateless and has a fixed address. Because of the fixed address, it does not need to be advertised among nodes, but we can only have a single translator within a 6LoCAN network.

3.1 Addressing Schema

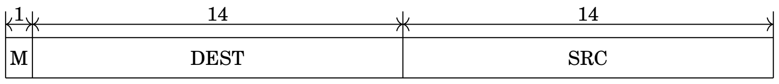


Figure 3.1: Address to Identifier Mapping

6LoCAN uses 14-bit node addresses to identify nodes on the bus. A node address has to be unique on the bus to avoid collisions. The Link-Layer duplicate address detection, defined in subsection 3.1.4, prevents node address collisions. The addressing schema describes how to map the 14-bit source and destination node-address to a 29-bit CAN identifier, as shown in Figure 3.1. The resulting CAN identifier is a combination of the source address, destination address, and a multicast bit. Because of the fact that a node address must be unique on the bus, the combination of the source and destination address always results in a unique identifier. This property is essential because collisions on the bus can only be resolved during the arbitration phase. If two nodes would send frames with the same identifiers, a collision in the data-phase could happen that cannot be resolved. Mapping only the destination address int the identifier would extend the address-space but leads to collision when two nodes try to send data to the same

Table 3.1: 6LoCAN address layout

Address	Description
0x3DFE - 0x3FFF	Reserved
0x3DFE	LLDAD
0x3DF1 - 0x3DFD	Reserved
0x3DF0	Ethernet Translator
0x0100 - 0x3DEF	Node addresses
0x0000 - 0x00FF	Reserved

receiver. Mapping only the source address to the identifier would not lead to collisions, but the nodes would need to inspect every packet, and the destination address would need to be included in every frame payload. With having the destination-node address as part of the identifier, the 6LoCAN node can use masked CAN filters to only receive frames dedicated to the node. This method reduces the number of interrupts for receiving and parsing frames. A dedicated multicast-bit marks frames as multicast frames. A node can use a masked CAN filter to receive multicast traffic from the attached multicast groups only.

Table 3.1 describes the address layout. Nodes can use any address from 0x0100 to 0x3DEF. Other addresses are either reserved or used for a particular purpose.

The conclusion of the addressing schema leads to the following properties:

- 14-bit address-space on the bus.
- The identifier uniquely identifies traffic from one node to another.
- The combination of source and destination address prevents collisions outside the arbitration-field.
- CAN filters that only pass relevant frames can easily be applied to the identifiers.

3.1.1 Unicast Address

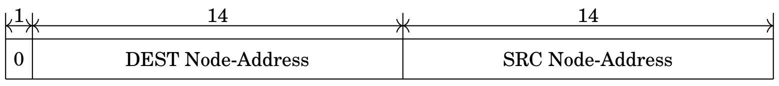


Figure 3.2: Unicast Node-Address to Identifier Mapping

Figure 3.2 depicts the mapping from source node-address and destination node-address to the CAN identifier for unicast traffic. The uppermost bit, the multicast bit, is set to zero. The destination node-address is located in the upper 14 bits, and the source node-address is located in the lower 14 bits.

3.1.2 Multicast Address

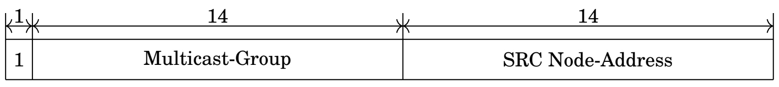


Figure 3.3: Multicast-Group to Identifier Mapping

Figure 3.3 depicts the mapping from source node-address and the multicast group to the CAN identifier for multicast traffic. The uppermost bit, the multicast bit, is set to one. The desired multicast group is located in the upper 14 bits, and the source node-address is located in the lower 14 bits. The multicast group corresponds to the lower 14 bits of the IPv6 multicast address. As shown in section 2.3 Figure 2.11, the Group-ID is located at the lower 112 bits of the IP address. Using the lower 14 bits is very efficient because all well-known multicast groups fit in that range. The node can also receive messages with a multicast Group-ID larger than 14 bits, but these messages may be ambiguous and need to be filtered afterward by the IPv6 address.

3.1.3 Address Generation

Every node is addressed by a 14-bit address as described in section 3.1. A node can either have a statically assigned address, or it can pick a random address whenever it connects to the bus or restarts the interface. In the case of a random address assignment, multiple nodes might attempt to claim the same address at the same time. Two nodes with the same address would be a violation of the requirement that every node must have a unique address on the bus. To prevent this situation, subsection 3.1.4 describes a mechanism to detect address duplications on the bus. A node that tries to claim an address already in use can then choose another address and rerun the detection for duplicate addresses.

Random address assignment allows nodes to join the network without any prior knowledge of the other nodes and without being provisioned first. It is especially useful for nodes that do not have an interface to enter an address or lack of none volatile memory. The drawback is that the address of a node may change their address whenever it reenters the network.

3.1.4 Link-Layer Duplicate Address Detection

To evaluate the need of the Link-Layer duplicate address detection lets first calculate the probability that two devices choose the same address, when they use an independent random number to assign their address. The address space a node can take its address from is 0x0100 to 0x3DEF (15599). Let us assume that we have 100 nodes on the bus. Depending on the transceivers used and wiring, this is a realistic electrical limit.

Let N be the number of possible addresses, n the number of nodes, and P the probability for a collision. The number of possible permutations of addresses and nodes is $N^n = 15599^{100}$.
The number of permutations without a collision is
$N \cdot (N-1) \cdot ... \cdot (N-(n-1))$.
Using the Laplace formula gives Equation 3.1.

$$P = 1 - \frac{N \cdot (N-1) \cdot \ldots \cdot (N-(n-1))}{N^n}$$
$$P = 1 - \frac{\prod_{i=1}^{n} N-(i-1)}{N^n} = 1 - \frac{n! \cdot \binom{N}{n}}{N^n} \tag{3.1}$$

$$P = 1 - \frac{100! \cdot \binom{15599}{100}}{15599^{100}} \approx 0.2724 \tag{3.2}$$

Equation 3.2 is the result of our example with 100 nodes.

The probability of having a collision is therefor 27.24%. For most applications, this number unacceptable, and a mechanism to prevent collisions needs to be applied.

For this reason, we introduced the Link-Layer duplicate address detection (LLDAD). LLDAD works on the Link-Layer only and utilizes a special CAN frame type, the Remote Transmission Request (RTR).

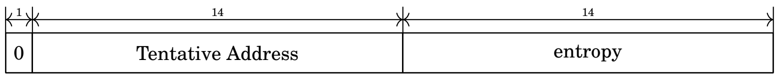


Figure 3.4: Link-Layer Duplicate Address Detection Request Frame

The collision detection begins with sending an LLDAD-request. The LLDAD-request is an RTR frame with no data. The 29-bit identifier has the same address-layout as a unicast 6LoCAN frame, but the source address is filled with entropy, and the destination address is filled with the tentative address as shown in Figure 3.4. The entropy prevents collisions in case another node tries to acquire the same address at the same time. The probability of not having a collision with 100 nodes, performing LLDAD at the same time, is calculated in Equation 3.3. The equation uses a slightly modified formula from Equation 3.1 with an address range extended by the 14-bit entropy.

$$P = \frac{100! \cdot \binom{15599 \cdot 2^{14}}{100}}{(15599 \cdot 2^{14})^{100}} \approx 0.99998 \tag{3.3}$$

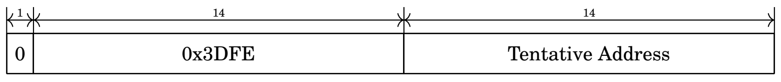


Figure 3.5: Link-Layer Duplicate Address Detection Response Frame

After the LLDAD-request is sent, the node waits at least 100ms for an LLDAD response frame. The response-frame is not an RTR frame and does not contain data. The identifier of the response is shown in Figure 3.5. The destination-address is the LLDAD response address 0x3DFE, and the source address is the address of the node that already owns the address, which is the same address as the tentative address. If there is no LLDAD-response frame with the tentative address as the source address on the bus, the tentative address is considered to be unique and becomes the address of the node. If the node receives an LLDAD-response, the LLDAD is failed, and in case the address was randomly chosen, it can retry with another random address. If the address was statically assigned, the node is not allowed to use the address and, therefore, cannot bring up the interface.

Figure 3.6 depicts the procedure of the LLDAD.

3.2 Stateless Address Autoconfiguration

The process of creating an IPv6 address is defined in subsection 2.3.5. However, forming the interface identifier depends on the underlying Link-Layer. The process of creating an interface identifier does not strictly follow the rules of RFC4291 [12] Appendix A. Instead of taking the 14-bit node address and zero fill it to the left, we use the method of creating an identifier from a 48-bit identifier. The reason for this is that this method was also chosen for 6LoWPAN (RFC4944 [19]) and the header compression is most efficient when the interface identifier if formed in that way.

For example, a node with the node-address of 0x1234 would create the interface identifier ::ff:fe00:1234/64.

Since the node address is only guaranteed to be unique on the bus-segment, the IPv6 Duplicate Address detection, as described in subsection 2.3.5,

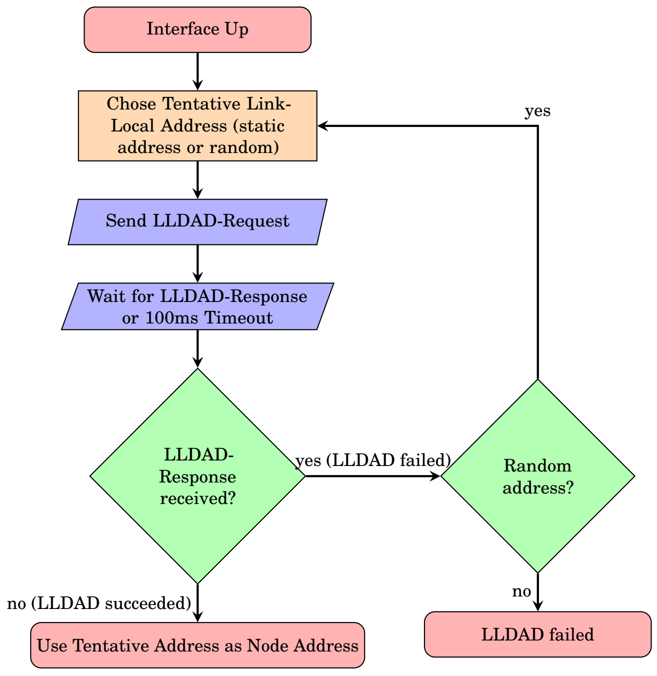

Interface Up
Chose Tentative Link-Local Address (static address or random)
Send LLDAD-Request
Wait for LLDAD-Response or 100ms Timeout
LLDAD-Response received?
yes (LLDAD failed)
Random address?
yes
no
no (LLDAD succeeded)
Use Tentative Address as Node Address
LLDAD failed

Figure 3.6: Link-Layer Duplicate Address Detection

needs to be performed. If the DAD failed, the node can either pick a new node-address or form an interface identifier that is not related to the node-address. The latter would result in an inefficient header compression and should, therefore, be avoided.

3.3 ISO-TP for 6LoCAN

IPv6 defines that the minimal MTU is 1280 bytes, but a CAN frame has a payload length of eight bytes for classical CAN and 64 bytes for CAN-FD. To satisfy the minimal MTU requirement, 6LoCAN uses a slightly modified version of ISO-TP. Some features like the address extension are not useful and, therefore, not used and not supported. The maximum packet size is limited to 4095 bytes, to limit the PCI of the first frame to two bytes. TSO-TP does only support unicast data transfer, but for IPv6, it is necessary to have multicast traffic too. Hence, we defined a way to have multicast transmissions, but without flow control.

3.3.1 Multicast

For multicast packets, we cannot have flow control, because there is more than one node receiving the packet and therefore, more than one node that would answer with a flow control frame. The sender cannot handle flow control for more than one receiving node because he does not know who and how many receivers there are.

Figure 3.7 shows the sequence of a multicast transfer. 6LoCAN defines that there has to be a pause between the First Frame and the first Consecutive Frame of at least 1ms. This pause allows all receiving nodes to set-up the CAN filter and allocate memory for the reception of the frame. The ST_{min} has to be chosen such that the slowest node on the bus can handle the reception. Multicast traffic is slower and maybe more unreliable than unicast transfers. For packets that fit in a Single Frame packet, multicast works the same way as unicast does.

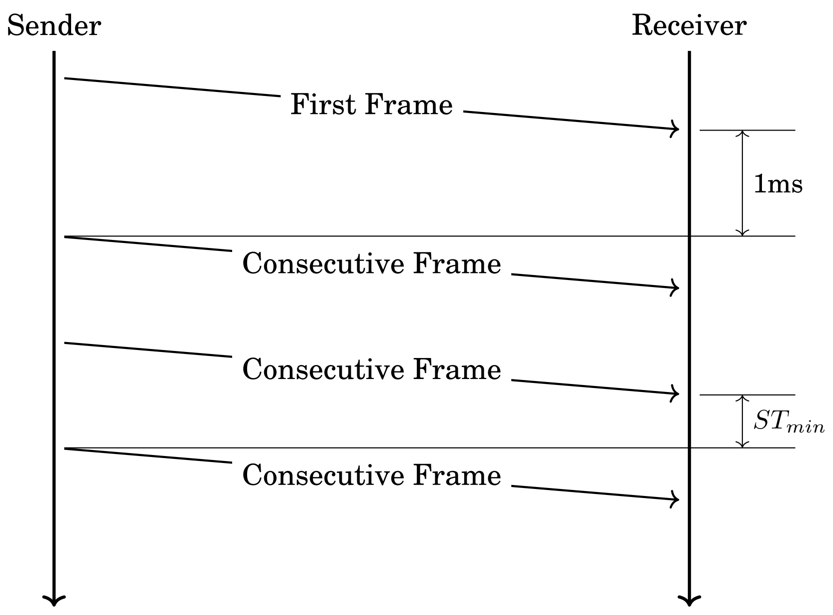


Figure 3.7: Example Multicast Sequence

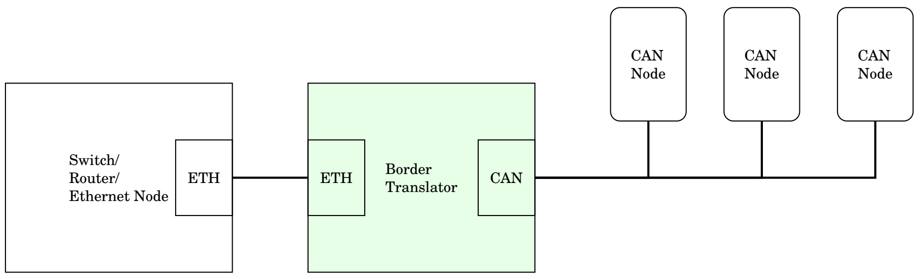


Figure 3.8: Ethernet Border Translator Schematic

3.3.2 Ethernet Border Translator

The Ethernet Border Translator is a concept for connecting a 6LoCAN bus segment to an Ethernet network. It is neither a router nor a switch. It only translates the packets from one Link-Layer to another, which means that the nodes stay in the same Link-Local domain. The advantage of this concept is that the translator is fully stateless and does not need any configuration and knowledge of the network. It allows us to use small devices with very little memory to be used as Border Translators. In case the 6LoCAN network segment needs a router, the Border Translator can be used to connect the 6LoCAN network to an ordinary Ethernet router. The Ethernet Border Translator can also be used to connect multiple 6LoCAN bus segments, using a standard Ethernet Switch. With the Ethernet Border Translator, we can use already existing Ethernet equipment for routing 6LoCAN traffic in the network. The advantage of keeping the devices in the same Link-Local domain is that multicast discovery protocols still works.

The translator has a fixed CAN node address (0x3DF0). With having a fixed address for the translator, the other nodes do not need a mechanism for discovering translators. 6LoCAN nodes do not need to know if a destination node the want to reach is in the same network, or connected via a translator. The sending node only needs to perform a neighborhood discovery as described in subsection 2.3.4. The border translator then forwards the neighbor solicitation message, and if the node is behind the translator, the answer will be forwarded by the translator. Since the address of the translator is well known, a node automatically knows that the packet is from outside the CAN bus segment, if the source node-address is the translator address. Packets forwarded from Ethernet to 6LoCAN carry the original 48 bit Ethernet MAC address directly after the ISO-TP First Frame header. Since the First Frame header has a size of two bytes, the inlined address always fits in the remaining frame data. A node receiving a packet from the border translator now replaces the node-address with the in-lined MAC address and saves it to the neighborhood cache. The node can now decide from the size of the address, stored in the neighborhood cache, if the packets need to be sent to the translator or to a node within the same bus-segment. Packets originating from a 6LoCAN node with a destination behind the translator are sent to the node-address of the translator and carry the destination Ethernet MAC-address inline. The border translator reassembles the packet, uncompresses the IPv6 header, and forwards the packet to the destination MAC-address. The source MAC-address is the 14-bit 6loCAN node-address extended by a 34-bit prefix. For packets from Ethernet to 6LoCAN, the translator compresses the IPv6 header and performs an ISO-TP transmission to the destination node. The destination node-address is taken from the last 14 bits of the destination MAC-address.

For neighborhood discovery packets, originating from the 6LoCAN network, with a Link-Layer address option (see subsection 2.3.4), the translator has to inspect the packet and extend the address in the option to match the extended Ethernet MAC-address.

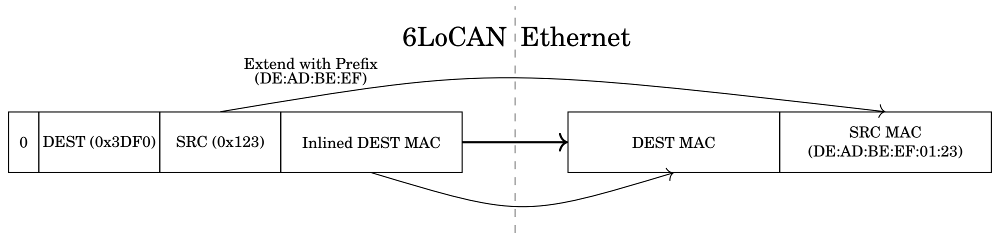


Figure 3.9: 6LoCAN to Ethernet Address Translation

Figure 3.9 shows how the 6LoCAN node address is extended to an Ethernet MAC-address and Figure 3.10 shows how the Ethernet MAC-address address is extended to a 6LoCAN node address.

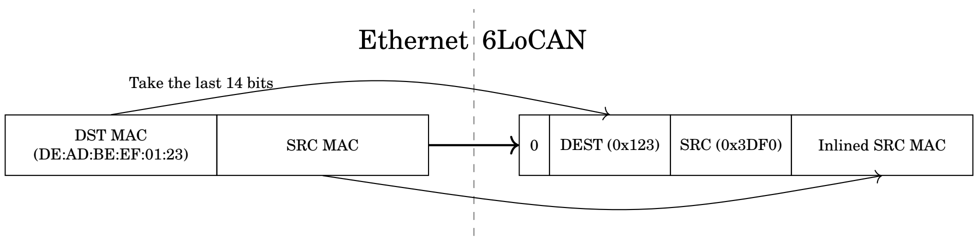


Figure 3.10: Ethernet to 6LoCAN Address Translation

4 Implementation

The 6LoCAN protocol is implemented into the Zephyr RTOS and made its way to the 2.0 release. The implementation was made in two phases. First we introduced a generic CAN API and an implementation of it for the STM32 microcontroller platform. In the second phase, I implemented the 6LoCAN translation layer into the Zephyr networking stack.

4.1 The Zephyr CAN API

Before this work, the Zephyr RTOS was lacking a CAN API, so the first step was to create a platform-agnostic API. The API should only support features defined by the CAN specification [5], to ensure that the network or application layer can rely on the API features.

The first API had an interface for the following features:

- Configure the mode and bitrate of the controller.
- Send a frame
- Attach a CAN filter and submit a callback whenever the filter matches on a frame
- Attach a CAN filter and submit the frame to a message queue whenever the filter matches
- Detach attached filters

It also defined containers for CAN frames and CAN filters called zcan_frame and zcan_filter. The containers provide a platform-agnostic way to handle frames and filters. The specific driver implementation is responsible for converting the zcan_frame and zcan_filter to the respective register values.

The Zephyr community agreed on my API proposal and the implementation for the STM32 platform and it was merged merged into Zephyr version 1.12. The API evolved over the time and got extended by an API the get the bus state and recover from bus-off state. Additionally people from the Zephyr community added two other driver implementations. One for an external CAN controller (MCP2515) and one for NXP platforms, based on NXPs flexcan IP.

4.1.1 Zephyr CAN frame

```
struct zcan_frame {
    u32_t id_type : 1;
    u32_t rtr     : 1;
    u32_t ext_id  : 29;
    u8_t  dlc;
    u8_t  data[8];
};
```

The struct above shows a simplified version of a Zephyr CAN frame. It is a container for a CAN frame and includes the identifier type, the identifier, a flag for RTR frames, the data length code, and the data data.

4.1.2 Zephyr CAN filter

```
struct zcan_filter {
    u32_t id_type   : 1;
    u32_t rtr       : 1;
    u32_t ext_id    : 29;

    u32_t rtr_mask    : 1;
    u32_t ext_id_mask : 29;
}
```

The struct above shows a simplified version of a Zephyr CAN filter struct. It is a container for a CAN filter and includes the identifier type, the identifier, the RTR bit, and the masks for the identifier and RTR-bit. The mask signals which bits of the identifier should be taken into account during the filtering. Bits that are set to one are compared and bits that are zero are ignored. Lets take an example of a filter with an identifier of 0x123 and a mask of

0xff. Frames with the identifier 0x123, 0x223, or generally 0xX23, where X is any number, would pass the filter. Frames with the identifier 0x124, 0x133, or generally 0xYXX, where XX is not 0x23, will not pass the filter.

4.1.3 Sending CAN frames

```
int can_send(struct device *dev, const struct zcan_frame *msg,
             s32_t timeout, can_tx_callback_t callback_isr,
             void *callback_arg);
```

The Send API is defined as shown above. It takes a pointer to the CAN device, a pointer to the frame container, a timeout, a callback function and an argument that is passed to the callback function. The function sends the frame as soon as a mailbox is ready to send a message. If a NULL pointer is passed to as the callback function, the call blocks until the message is sent. Otherwise the callback is called when the message is sent.

4.1.4 Receiving CAN frames

```
int can_attach_isr(struct device *dev,
                   can_rx_callback_t isr, void *callback_arg,
                   const struct zcan_filter *filter);
```

The main receiving function is shown above. It takes a pointer to the CAN device, the function that should be called when a frame that passes the filter is received, an argument that is passed to the callback and the CAN filter. The function returns a filter-id to uniquely identify the attached filter. The filter-id is used for detaching the filter when needed. The CAN controller reads any message on the bus and compares the identifier to the attached filters. If a filter matches, the respective callback is called. If the identifier matches more that one filter, the function called depends on the metrics of the CAN controller. The callback is called in an interrupt context and therefor is not allowed to block and should keep the execution time as low as possible.

```
int can_attach_workq(struct device *dev, struct k_work_q *work_q,
                     struct zcan_work *work,
                     can_rx_callback_t callback,
                     void *callback_arg,
                     const struct zcan_filter *filter);
```

The can_attach_workq function is a wrapper for the can_attach_isr, that calls the callback function in the context of the provided work queue instead of an ISR context.

```
int can_attach_msgq(struct device *dev, struct k_msgq *msg_q,
                    const struct zcan_filter *filter);
```

The can_attach_msgq function attaches a message queue instead on a callback. Whenever a frame that matches the filter is received, it is put into the message queue. The message queue can then be read within a thread.

4.2 6LoCAN Implementation

A Link-Layer Implementation in Zephyr consist of two layers. The Network Device Driver and the Link-Layer implementation (L2), which are highlighted in Figure 4.1. For the 6LoCAN implementation, the Network Device Driver is an abstraction of the Zephyr CAN API that installs the respective frame filters, handles the incoming CAN frames in an interrupt context, puts them into network packets, sets the Link-Layer address of them, and and hands the packet over to the receiving work-queue. For sending CAN frames, the abstraction only forwards the raw CAN frames. The 6LoCAN Link-Layer implementation does the parsing of the ISO-TP header, fragmentation and reassembly, and IPHC header compression and decompression. The device driver implementation is explained in detail in subsection 4.2.1, and the L2 implementation is explained in detail in subsection 4.2.2.

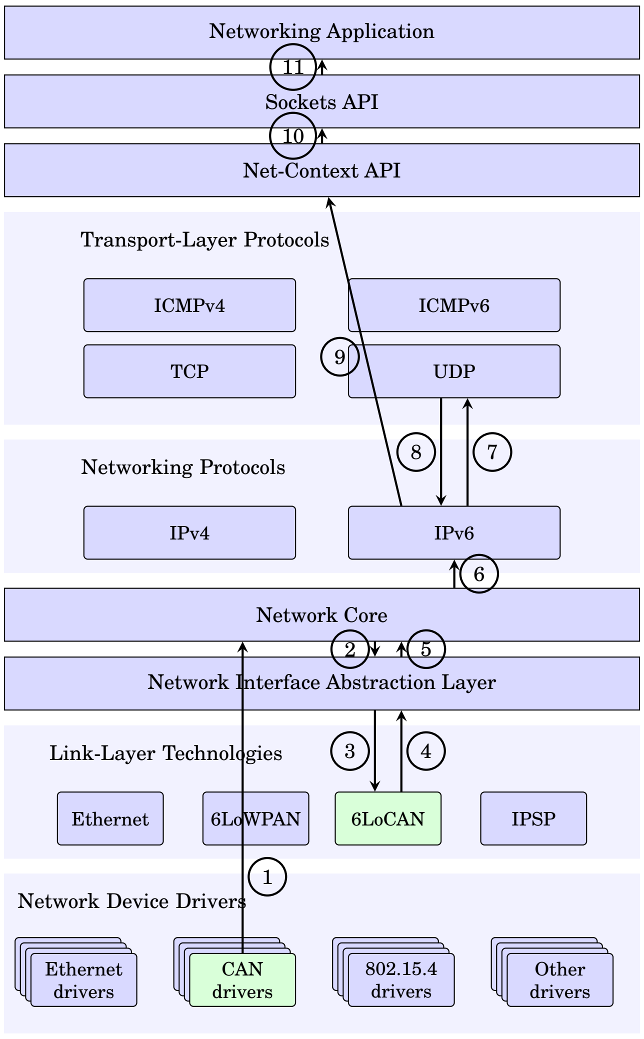


Figure 4.1: Zephyr Network Stack 6LoCAN RX example

Figure 4.1 shows an example of receiving a UDP packet.

1. The CAN network device driver uses the CAN API to receive the raw CAN frames. The frames are copied into network packets and handed over to the Net Core. The Net Core accepts the packet and puts it into a receiving work-queue, depending on the priority of the packet.
2. When the receiving queue is scheduled, the packet is handed over to the Network Interface.
3. The Network Interface calls the 6LoCAN L2 implementation with the received packet, which analyzes the frame content. The L2 implementation performs the reassembly (ISO-TP) and when the packet is complete, the IPHC decompression.
4. If the packet is complete and decompression is successful, the reassembled packet is passed back to the Network Interface.
5. The packet is then forwarded to the Net Core.
6. The Network Core checks the IP header version and if it is IPv6, the IPv6 implementation is called. The IPv6 implementation checks the IPv6 headers and discards the packet if the destination does not match any addresses of the interface.
7. Depending on the next header field, the specific transport layer implementation is called. In this Example, it is UDP.
8. The UDP implementation parses the UDP header, checks the checksum and length of the packet, and returns the result to the IPv6 implementation.
9. If the packet is valid, the IPv6 implementation passes the packet to the Net-Context API.
10. The Net-Context API checks for registered sockets on the destination-port.
11. If a socket is listening to the destination-port, the Socket-API passes the packet to the networking-application.

Figure 4.2 shows an example of sending a UDP packet.

1. The Application sends a chunk of data by using the Socket-API.
2. The Socket-API binds the request to the context of the opened socket and uses the API to send the data.
3. The Net-Context implementation calls the IPv6 implementation to fill the IPv6 headers. The interface information, like the source IP address, is taken from the interface associated to the Net-Context.
4. The Net-Context implementation calls the UDP implementation to fill the UDP headers.
5. The packet is then handed over to the network core.
6. The network core calls the Network Interface abstraction, which fills the Link-Layer addresses (from either neighborhood discovery or cache).
7. The Network Interface abstraction then puts the packet into the sending work-queue. The work-queue thread then calls th 6LoCAN Link-Layer. The Link-Layer creates a ISO-TP context and performs the fragmentation.
8. For every frame that is sent, the CAN network driver is called.

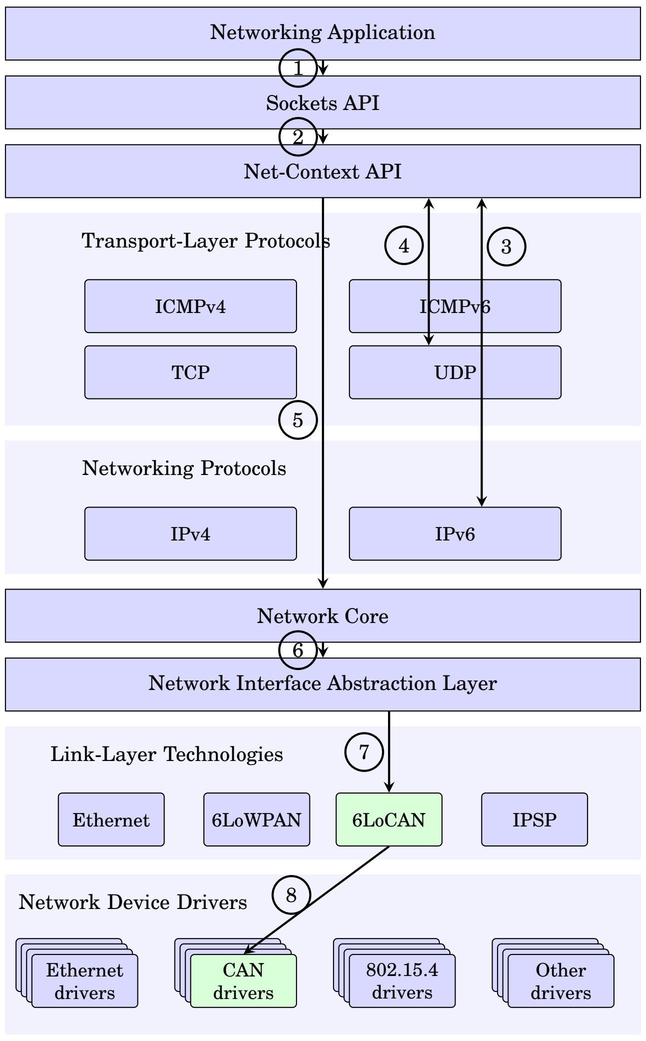


Figure 4.2: Zephyr Network Stack 6LoCAN TX example

4.2.1 6LoCAN Network Device Driver

The 6LoCAN Network Device Driver is responsible for receiving and sending RAW CAN frames. For this purpose it is using the Zephyr CAN API. The API exposes five functions:

- Interface Initialization
- Enabling/Disabling
- Sending
- Attaching Filters
- Detaching Filters

Interface Initialization binds the driver context to an interface, calls the initialization function of the 6LoCAN Link-Layer and registers the multicast monitor. The multicast monitor is called whenever a multicast group is added to an interface.

Enable is called from the 6LoCAN Link-Layer whenever the interface changes its state from up (enabled) to down (disabled) or vice versa. If the interface is going to be brought up, then this function attaches the unicast CAN filter. This filter receives unicast frames where the destination address matches the interface link-local address. If the interface is going to be brought to the down state, the unicast filter is detached.

Send forwards the incoming frames to the Zephyr CAN API without any modification.

Attach Filter and **Detach Filter** are wrappers for the Zephyr CAN API, and work the same way the Zephyr CAN API do. They are used to attach filters for the Link-Layer Duplicate Address Detection (LLDAD), from the 6LoCAN Link-Layer.

Whenever a frame on the bus matches the unicast filter, the receiving function is called. This function allocates a net packet that is large enough to hold the frame payload, the source and the destination Link-Layer address. The addresses are taken from the frame identifier and copied into the packet, followed by the frame payload. The resulting packet is then put into the receiving work-queue of the network stack, which later passes it to the 6LoCAN Link-Layer.

The multicast monitor is responsible for attaching filters for the corresponding multicast group. The filter matches the the last 14 bits of the IP address and the multicast flag. The source address is ignored for the comparison. For receiving the frames, the same receiving function is used, as it is used for unicast frames.

4.2.2 6LoCAN Link-Layer

The 6LoCAN Link-Layer gets the raw CAN frames from the 6LoCAN Network Device Driver in the thread context of the receiving queue. It is responsible for handling the ISO-TP transfers that can either be a single frame or a fragmented transfer consisting of many frames. If the packet does not fit into a single frame, the fragmentation, reassembly and flow control is also managed by this layer. Fragmented packets are associated with a context that represents a single ISO-TP transfer. This context contains all the information necessary to handle the segmentation, reassembly and flow control. The LLDAD is also performed from this layer whenever the interface is initialized.

Whenever a frame arrives, the ISO-TP PCI type is checked to determine what to do next with the frame.

Single frames can be handled in one shot and therefor need no additional context. The implementation removes the ISO-TP header, checks the length and performs the IPHC decompression. After that, the packet is processed by the higher layers.

If a First Frame arrives, the implementation reads the total packet length and allocates a new packet that is able to carry the reassembled data. To keep track of the reassembly process, an ISO-TP context is allocated and linked with the newly allocated packet. The context contains the actual state of reassembly, timeout timers, a remaining data counter, sequence number and the block counter. The context is then initialized with the data from the first frame, the residual data is copied to the packet and a if the it is not a multicast transfer, a Frame Control Frame is sent back to the sender.

For Consecutive frames, the implementation checks if is a context that is in the state of receiving consecutive frames and is linked to a packet from the same sender. If a context could be found, the data in the Consecutive Frame is added to the packet. If the counter for remaining data in this context reaches zero, the reassembled packet is pushed to the receiving queue and ands up in this implementation again. On the finished packet, IPHC decompression is performed and the upper layers continue processing the packet.

Outgoing packets can either be single frames, if they are short enough, or need to be split into several frames. The frame Identifier is composed from the Link-Layer addresses. The Link-Layer source and destination address is already defined by the packet. If the packet can be sent as a single frame, the implementation creates a CAN frame with an ISO-TP Single Frame header and the data from the packet. If the packet needs several frames, the implementation allocates an ISO-TP sending context, initializes it with the information from the packet, and sends out the ISO-TP first frame. The implementation is then waiting for a Frame Control Frame to continue with sending the Consecutive frames. The implementation sends as many consecutive frames as necessary to complete the transfer.

5 Evaluation

For evaluation purposes, 6LoCAN is implemented in the Zephyr RTOS network stack.

Table 5.1: Resource demand of 6LoCAN

Layer	RAM	ROM
CAN Driver	256	3878
Network CAN Driver	328	1354
IPHC	4	2779
6LoCAN	1664	6534
Networking (incl. 6LoCAN, without drivers)	34016	57269

Table 5.1 shows the RAM and ROM demand of the 6LoCAN implementation. The build is the echo_server sample application, built on Zephyr 2.2 with 6LoCAN enabled. It includes the IPv6 network stack, TCP, and UDP. The sample uses 64 net buffers for each, receiving and sending. One buffer can hold up to 128 bytes, which results in 8192 bytes for sending and 8192 bytes for sending. As shown in the table, the 6LoCAN does not add an excessive amount of RAM and ROM. The sample uses 64 net buffers each, for receiving and sending. As shown in the table, the 6LoCAN does not add an excessive amount of RAM and ROM. Most of the RAM is used for the sending and receiving buffers. The 6LoCAN contexts use 1024 of the 1664 bytes of RAM. The ROM overhead of 6LoCAN in the entire network stack is only 11.4 %.

Figure 5.1 shows the setup used for all evaluated data in this section. USB is used as a power supply and provides a UART terminal. The boards are flashed with an application that transfers data when issuing a command over UART. The boards are connected to the CAN-bus with a CAN-

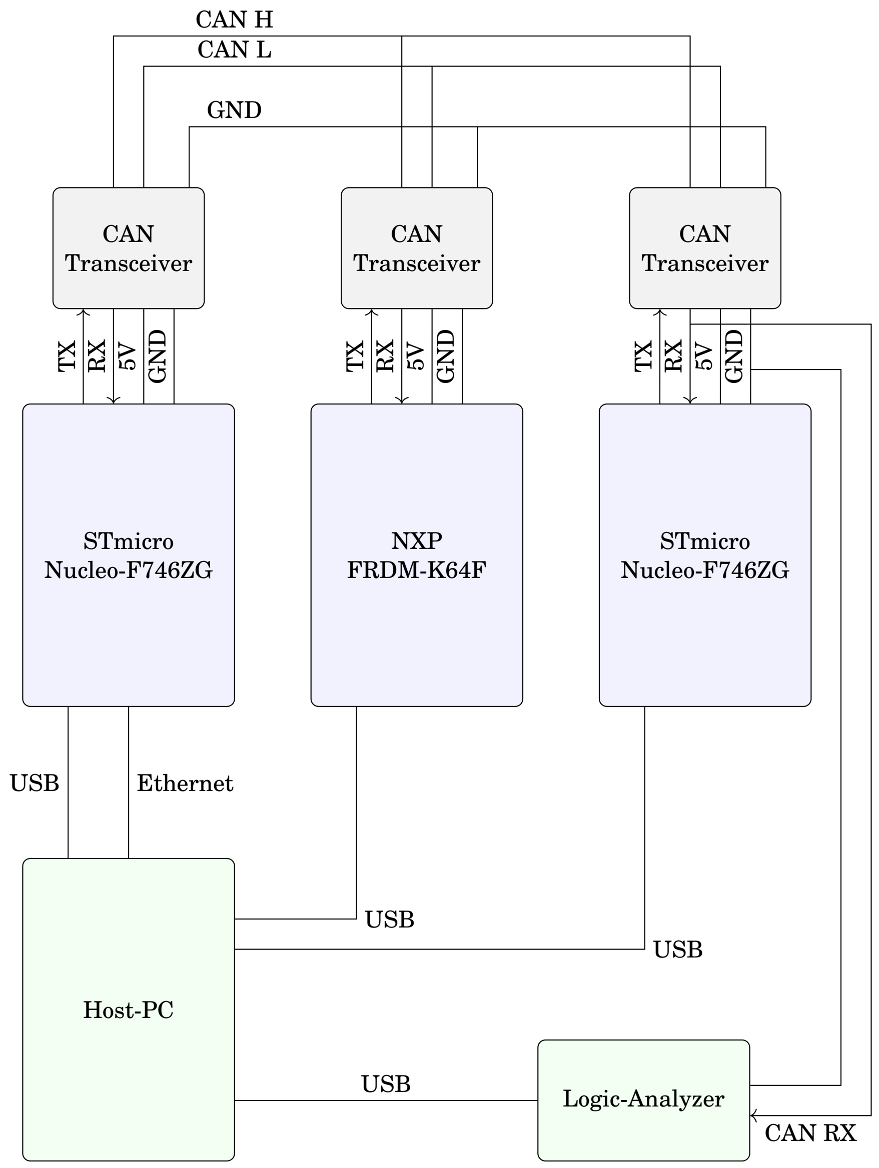


Figure 5.1: Test Setup

transceiver. One board also has Ethernet, connected to the host. This Ethernet port serves as the Boarder-Translator. The Logic-Analyzer is used to determine the exact timing of the frames.

5.0.1 Link-Layer Duplicate Address Detection

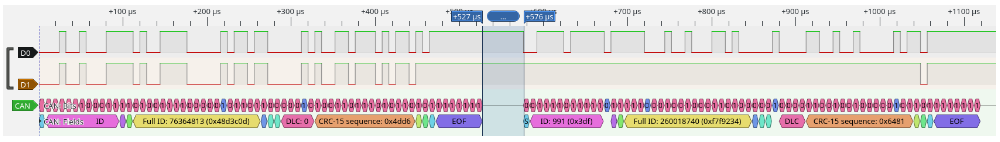


Figure 5.2: Link-Layer Duplicate Address Detection Measurement

Two boards are programmed with the same fixed address (0x1234) to evaluate the Link-Layer Duplicate Address Detection. Figure 5.2 shows a capture of the LLDAD. The black trace is the CAN_RX line and therefor shows the logic levels on the bus. The brown trace is the CAN_TX line and, therefore, only shows the data sent by the node that issues the LLDAD. The first CAN frame is the LLDAD request. The identifier used is 0x48d3c0d (DST: 0x1234, SRC (entropy): 0x3C0D) and the RTR bit is set. The second frame is the response from the other node, already using the same address. The identifier is 0xf7f9234 (DST: 3DEF, SRC: 0x1234). The node that caused the address duplication noticed the duplication and disabled the interface.

```
[00.001] <inf> net_l2_canbus: DAD failed

Interface 0x2002dde0 (CANBUS) [1]
=================================
Interface is down.
```

5.0.2 Ping

This section shows the evaluation of a Ping command. One board has a fixed address of 0x1234, and the other board has a randomly chosen address. The logs below show what is going on during the brin up of the interface and pings.

First node with the address 0x1234:

```
================================== Interface bring up ==================================
[00.471] Sending  Multicast Listener Report v2 type 143 code 0 from :: to ff02::16
[00.471] Sending  Multicast Listener Report v2 type 143 code 0 from :: to ff02::16
[00.471] Sending  Neighbor Solicitation type 135 code 0 from :: to ff02::1:ff00:1234 <—(DAD)
[00.472] Sending  Router Solicitation   type 133 code 0 from :: to ff02::1
[01.472] Sending  Router Solicitation   type 133 code 0 from fe80::ff:fe00:1234 to ff02::1
[01.479] Router Solicitation received   type 133 code 0 from fe80::ff:fe00:314e to ff02::1
[02.472] Sending  Router Solicitation   type 133 code 0 from fe80::ff:fe00:1234 to ff02::1
[02.479] Router Solicitation received   type 133 code 0 from fe80::ff:fe00:314e to ff02::1
=========================================================================================

====================== Start of the Neighbor Dicovery from the pinging node ========================
[03.577] Neighbor Solicitation received   type 135 code 0 from fe80::ff:fe00:314e to ff02::1:ff00:1234
[03.577] Sending  Neighbor Solicitation   type 135 code 0 from fe80::ff:fe00:1234 to ff02::1:ff00:314e
[03.577] Sending Neighbor Advertisement   type 136 code 0 from fe80::ff:fe00:1234 to fe80::ff:fe00:314e
[03.600] Neighbor Solicitation received   type 135 code 0 from fe80::ff:fe00:314e to ff02::1:ff00:1234
[03.600] Sending Neighbor Advertisement   type 136 code 0 from fe80::ff:fe00:1234 to fe80::ff:fe00:314e
[03.609] Neighbor Advertisement received type 136 code 0 from fe80::ff:fe00:314e to fe80::ff:fe00:1234
=====================================================================================================

========================================== Pings ==========================================
[03.618] Echo Request received type 128 code 0 from fe80::ff:fe00:314e to fe80::ff:fe00:1234
[03.618] Sending Echo Reply    type 129 code 0 from fe80::ff:fe00:1234 to fe80::ff:fe00:314e
[04.573] Echo Request received type 128 code 0 from fe80::ff:fe00:314e to fe80::ff:fe00:1234
[04.574] Sending Echo Reply    type 129 code 0 from fe80::ff:fe00:1234 to fe80::ff:fe00:314e
[05.574] Echo Request received type 128 code 0 from fe80::ff:fe00:314e to fe80::ff:fe00:1234
[05.574] Sending Echo Reply    type 129 code 0 from fe80::ff:fe00:1234 to fe80::ff:fe00:314e
===========================================================================================
```

Second node with the address 0x314e:

```
================================== Interface bring up ==================================
[00.101] Sending Multicast Listener Report v2 type 143 code 0 from :: to ff02::16
[00.102] Sending Multicast Listener Report v2 type 143 code 0 from :: to ff02::16
[00.102] Sending Neighbor Solicitation type 135 code 0 from :: to ff02::1:ff00:314e
[00.102] Sending Router Solicitation   type 133 code 0 from :: to ff02::1
[01.103] Sending Router Solicitation   type 133 code 0 from fe80::ff:fe00:314e to ff02::1
[01.107] Router Solicitation received  type 133 code 0 from fe80::ff:fe00:1234 to ff02::1
[02.103] Sending Router Solicitation   type 133 code 0 from fe80::ff:fe00:314e to ff02::1
[02.107] Router Solicitation received  type 133 code 0 from fe80::ff:fe00:1234 to ff02::1
=========================================================================================

======================================= First Ping =======================================
[03.198] Sending Echo Request type 128 code 0 from fe80::ff:fe00:314e to fe80::ff:fe00:1234
==========================================================================================

====== Start of the Neighbor Dicovery to find out the Link-Layer address of fe80::ff:fe00:1234 =======
[03.199] Sending Neighbor Solicitation    type 135 code 0 from fe80::ff:fe00:314e to ff02::1:ff00:1234
[03.215] Neighbor Solicitation received   type 135 code 0 from fe80::ff:fe00:1234 to ff02::1:ff00:314e
[03.216] Sending Neighbor Solicitation    type 135 code 0 from fe80::ff:fe00:314e to ff02::1:ff00:1234
[03.216] Sending Neighbor Advertisement   type 136 code 0 from fe80::ff:fe00:314e to fe80::ff:fe00:1234
[03.224] Neighbor Advertisement received type 136 code 0 from fe80::ff:fe00:1234 to fe80::ff:fe00:314e
[03.245] Neighbor Advertisement received type 136 code 0 from fe80::ff:fe00:1234 to fe80::ff:fe00:314e
=====================================================================================================
======================== First Ping reply and remaining two Pings =======================
[03.253] Echo Reply received  type 129 code 0 from fe80::ff:fe00:1234 to fe80::ff:fe00:314e
[04.199] Sending Echo Request type 128 code 0 from fe80::ff:fe00:314e to fe80::ff:fe00:1234
[04.208] Echo Reply received  type 129 code 0 from fe80::ff:fe00:1234 to fe80::ff:fe00:314e
[05.200] Sending Echo Request type 128 code 0 from fe80::ff:fe00:314e to fe80::ff:fe00:1234
[05.209] Echo Reply received  type 129 code 0 from fe80::ff:fe00:1234 to fe80::ff:fe00:314e
=========================================================================================
```

The Ping command:

```
uart:~$ net ping fe80::ff:fe00:1234
PING fe80::ff:fe00:1234
8 bytes from fe80::ff:fe00:1234 to fe80::ff:fe00:314e: icmp_seq=0 ttl=64 time=54 ms
8 bytes from fe80::ff:fe00:1234 to fe80::ff:fe00:314e: icmp_seq=1 ttl=64 time=9 ms
8 bytes from fe80::ff:fe00:1234 to fe80::ff:fe00:314e: icmp_seq=2 ttl=64 time=9 ms
```

The first ping takes 45 ms longer than the following two pings. This additional time is due to the Neighbor Discovery (ND). The nodes first have to discover the Link-Layer addresses before they can exchange the data.

Table 5.2: UDP data throughput and protocol overhead

Payload	BS	STmin	IPv6	ISO-TP	Frames	Measured
bytes	Block Size	ms	bytes	bytes	#	ms
128	0	0	8	24	21	22.63
128	8	0	8	30	23	24.66
128	8	5	8	30	23	105.17
1024	0	0	8	152	149	161.24

Table 5.2 shows the protocol overhead and the time needed to send UDP data packets. The IPv6 row shows the overhead coming from the IPv6 protocol, and the ISO-TP row shows the overhead coming from the ISO-TP protocol. If the data would be transferred using raw ISO-TP, eight bytes could be saved. This value is constant, regardless of the packet-size. If the data would be transferred using raw CAN frames, it would take 16 frames to send 128 bytes and 128 frames for 1024 bytes.

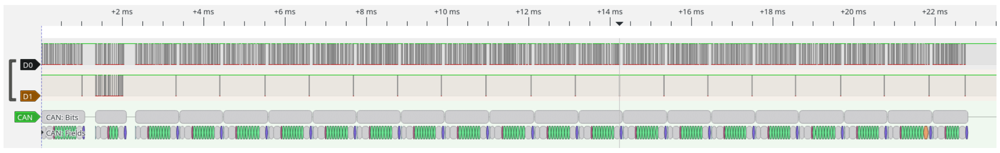


Figure 5.3: UDP Transfer 128 byte, BS=0, STmin=0

Figure 5.3is a capture of an UDP transfer with 128 bytes. It uses the fastest possible parameter set with a BS of zero, which means that there are no additional FC frames to wait for and no separation time.

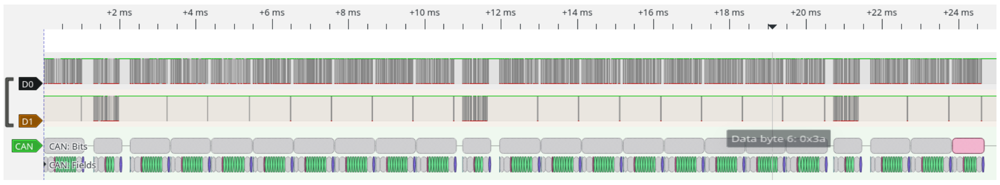


Figure 5.4: UDP Transfer 128 byte, BS=8, STmin=0

Figure 5.4 is a capture of an UDP transfer with 128 bytes. It uses a BS of eight, which means that every eight frames, the sender has to wait for another FC frame with a CTS state. The separation time is zero.

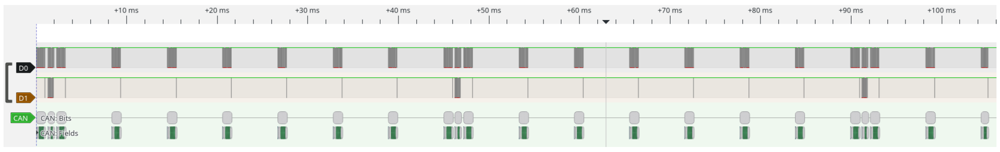


Figure 5.5: UDP Transfer 128 byte, BS=8, STmin=5

Figure 5.5 is a capture of an UDP transfer with 128 bytes. It shows a combination of a BS of eight and a separation time of five ms. The separation time can, for example, be used to reduce the load on the bus.

6 Conclusion

6LoCAN brings end-to-end IPv6 to small-scale microcontrollers with only a few additional components, like the CAN transceiver. It can be used for non-realtime communication and bulky data transfers. If configured to use a random address assignment, it is zero-configuration capable and does not require any settings or persistent memory. Nodes can join a 6LoCAN network seamlessly. The nodes could, for example, be identified with host-names and multicast DNS discovery. The CAN-bus is not the most efficient bearer for IPv6 traffic, because of the small frame payload of eight bytes for classic CAN, but with the 64-byte payload of CAN-FD, packets with small payload can even fit a single frame. By using IPv6 instead of raw data transfers, it is possible to use any application-layer protocol that works on top of IP. With IPv6, it is, for example, possible to encrypt the traffic using the well known TLS protocol [20]. Devices that already have native IPv6 support can easily be connected to large-scale networks like the internet. 6LoCAN supports multicast groups natively, and therefore, very efficiently. Packets to dedicated groups are only received by nodes that subscribed to the group. This property could be used for efficient implementations of a publisher and subscriber models like MQTT. Since 6LoCAN only uses the 29-bit extended addresses, the 11-bit standard address range is still usable for other protocols. An 11-bit standard address always has a higher priority on the bus than an extended address, and therefore the standard address-range can still even be used for high priority traffic, like real-time events. It could also share the bus with other existing protocols like CANopen. CANopen could, for example, be used for machine control, as ith has been used before, and 6LoCAN could be used for interfacing control panels.

The 6LoCAN standard-draft [25] was submitted to the Internet Engineering Task Force (IETF) in October 2019.

Bibliography

[1] Zigbee Alliance. *Project Connected Home over IP*. https://www.connectedhomeip.com/. Feb. 2020.

[2] CAN in Automation. *CANopen – The standardized embedded network*. https://www.can-cia.org/canopen/. Feb. 2020.

[3] Fred Baker et al. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. RFC 2474. Dec. 1998. DOI: 10.17487/RFC2474. URL: https://rfc-editor.org/rfc/rfc2474.txt.

[4] Robert T. Braden. *Requirements for Internet Hosts - Communication Layers*. RFC 1122. Oct. 1989. DOI: 10.17487/RFC1122. URL: https://rfc-editor.org/rfc/rfc1122.txt.

[5] *CAN Specification 2.0*. Specification. Stuttgart, DE: Robert Bosch GmbH, Sept. 1991.

[6] *CAN with Flexible Data-Rate*. Specification. Gerlingen, DE: Robert Bosch GmbH, Apr. 2011.

[7] Dr. Steve E. Deering and Bob Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 8200. July 2017. DOI: 10.17487/RFC8200. URL: https://rfc-editor.org/rfc/rfc8200.txt.

[8] Sally Floyd, Dr. K. K. Ramakrishnan, and David L. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. Sept. 2001. DOI: 10.17487/RFC3168. URL: https://rfc-editor.org/rfc/rfc3168.txt.

[9] Eclipse Foundation. *Eclipse Sparkplug Working Group*. https://sparkplug.eclipse.org/. Feb. 2020.

[10] Mukesh Gupta and Alex Conta. *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*. RFC 4443. Mar. 2006. DOI: 10.17487/RFC4443. URL: https://rfc-editor.org/rfc/rfc4443.txt.

[11] Bob Hinden and Dr. Steve E. Deering. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. Dec. 1998. DOI: 10.17487/RFC2460. URL: https://rfc-editor.org/rfc/rfc2460.txt.

[12] Robert M. Hinden and Stephen E. Deering. *IP Version 6 Addressing Architecture*. RFC 4291. Feb. 2006. DOI: 10.17487/RFC4291. URL: https://rfc-editor.org/rfc/rfc4291.txt.

[13] IEEE and the Open Group. "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7." In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (Jan. 2018), pp. 1–3951. ISSN: null. DOI: 10.1109/IEEESTD.2018.8277153.

[14] *Internet Protocol*. RFC 791. Sept. 1981. DOI: 10.17487/RFC0791. URL: https://rfc-editor.org/rfc/rfc791.txt.

[15] *Road vehicles — Controller area network. Part 1: Data link layer and physical signalling*. Standard. Geneva, CH: International Organization for Standardization, Dec. 2003.

[16] *Road vehicles — Diagnostic communication over Controller Area Network (DoCAN). Part 2: Transport protocol and network layer services.* Standard. Geneva, CH: International Organization for Standardization, Dec. 2016.

[17] M. Jung, C. Reinisch, and W. Kastner. "Integrating Building Automation Systems and IPv6 in the Internet of Things." In: *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. July 2012, pp. 683–688. DOI: 10.1109/IMIS.2012.134.

[18] Seiichi Kawamura and Masanobu Kawashima. *A Recommendation for IPv6 Address Text Representation*. RFC 5952. Aug. 2010. DOI: 10.17487/RFC5952. URL: https://rfc-editor.org/rfc/rfc5952.txt.

[19] Gabriel Montenegro et al. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944. Sept. 2007. DOI: 10.17487/RFC4944. URL: https://rfc-editor.org/rfc/rfc4944.txt.

[20] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://rfc-editor.org/rfc/rfc8446.txt.

[21] William A. Simpson et al. *Neighbor Discovery for IP version 6 (IPv6)*. RFC 4861. Sept. 2007. DOI: 10.17487/RFC4861. URL: https://rfc-editor.org/rfc/rfc4861.txt.

[22] *Controller Area Network Physical Layer Requirements*. Application Report. Texas Instruments, Jan. 2008.

[23] Susan Thomson, Thomas Narten, and Tatuya Jinmei. *IPv6 Stateless Address Autoconfiguration*. RFC 4862. Sept. 2007. DOI: 10.17487/RFC4862. URL: https://rfc-editor.org/rfc/rfc4862.txt.

[24] Pascal Thubert and Jonathan Hui. *Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks*. RFC 6282. Sept. 2011. DOI: 10.17487/RFC6282. URL: https://rfc-editor.org/rfc/rfc6282.txt.

[25] Alexander Wachter. *IPv6 over Controller Area Network*. Internet-Draft draft-wachter-6lo-can-00. Work in Progress. Internet Engineering Task Force, Oct. 2019. 17 pp. URL: https://datatracker.ietf.org/doc/html/draft-wachter-6lo-can-00.