Introduction to C++ with Linux CAN

Austin Yang ∈ Illini RoboMaster
[austiny2@illinois.edu, https://www.illinirobomaster.com]

September 28, 2025

Contents

1 Introduction

1(a) Plan

This tutorial will go through can.h and can.cc line-by-line as an introduction to C++. We will cover concepts in:

- Basic C++ syntax and structure
- Object-oriented programming concepts
- Libraries and the standard library
- GNU+Linux and UNIX

You can find the files covered at
https://github.com/illini-robomaster/irm_jetson/blob/main/src/include/board/can.h
and the source file code at
https://github.com/illini-robomaster/irm_jetson/blob/main/src/include/board/can.cc
You can find the examples at
https://github.com/illini-robomaster/irm_jetson/blob/main/src/examples/can_recieve.cc
https://github.com/illini-robomaster/irm_jetson/blob/main/src/examples/can_send.cc
https://github.com/illini-robomaster/irm_jetson/blob/main/src/examples/motor_m3508.cc

1(b) Basic Knowledge

I will assume the most basic knowledge of programming. I will take time in the introduction to go over types and pointers. If you are familiar, please skip this subsection and proceed directly to section 2.

Do note that these two introductions are partially written by AI.

1(b).1 Types

In C++, a type is a classification that specifies what kind of value a variable can hold and what operations can be performed on it. C++ has several fundamental types: C++ also

Type	Description	Size (typical)
`bool`	Boolean value	1 byte
`char`	Character	1 byte
`int`	Integer	4 bytes
`float`	Single-precision floating point	4 bytes
`double`	Double-precision floating point	8 bytes
`void`	No type	N/A

Table 1: Common C++ Fundamental Types

allows for type modifiers like `signed`, `unsigned`, `short`, and `long` to modify the range

of values a type can hold. Additionally, C++ supports compound types like arrays, pointers, and references.

In our code, we will use the more verbose types like `uint8_t` etc. This is common in embedded programming.

1(b).2 Pointers

A pointer is a variable that stores the memory address of another variable. Think of it as a label that points to a location in memory where data is stored. Pointers are fundamental to C++ and enable features like dynamic memory allocation and efficient array handling. Here's a simple example:

Operator	**Symbol**	**Purpose**
Address-of	`&`	Gets the memory address of a variable
Dereference	`*`	Accesses the value at the memory address
Member access	`->`	Accesses members of an object through a pointer

Table 2: Pointer Operators in C++

```
int x = 5;  // Declare an integer variable
int *ptr;  // Declare a pointer to an integer
ptr = &x;  // Store the address of x in ptr
```

After this code executes, `ptr` contains the memory address of `x`. To access the value stored at that address (i.e., the value of `x`), you would dereference the pointer using `*ptr`, which would give you `5`. Pointers are especially important in C++ for memory management,

Concept	**Syntax**	**Explanation**
Pointer declaration	`int *ptr`	Declares a pointer to an integer
Address assignment	`ptr = &x`	Assigns the address of `x` to `ptr`
Dereferencing	`*ptr = 10`	Sets the value at the address stored in `ptr` to 10
Pointer to pointer	`int **ptr2`	A pointer to a pointer to an integer

Table 3: Pointer Syntax and Usage

creating dynamic data structures, and interfacing with system-level code.

2 Understanding our CAN Header File

We will start by examining our header file can.h.

2(a) Why header and source files?

In C++, we typically split our code into *header* files (`.h`) and *source* files (`.cc` or `.cpp`) for two main reasons:

Convenience: Header files act as an interface for our code. They tell other programmers (and the compiler) what functions and classes are available, what parameters they take, and what they return. This makes it easier for others (humans or your IDE) to work with your code.
Compilation: When you `#include` a file, you are essentially pasting its contents into place. You don't want the implementation there.

The compilation process works in two main stages:

1. Compile: Each `.cc` file is compiled independently into an object file (`.o`), checking what functions and classes exist/their definitions against the header files (`.h`). *Preprocessor directives* (code starting with `#`) are run **before** compilation.

2. Link: The *linker* combines all the object files together, resolving references between them to create the final executable.

Remark | If you were to write foo.h without the corresponding implementation in foo.cc and tried to `#include` it in another file, you would only get a complaint during the linking step (you would see the `ld` (linker) program error).

2(b) Include Guards and Headers

```
1  /*******************************************************************************
2   *                                                                             *
3   *  Copyright (C) 2025 RoboMaster.                                             *
4   *  Illini RoboMaster @ University of Illinois at Urbana-Champaign             *
5   *                                                                             *
6   *  This program is free software: you can redistribute it and/or modify       *
7   *  it under the terms of the GNU General Public License as published by       *
8   *  the Free Software Foundation, either version 3 of the License, or          *
9   *  (at your option) any later version.                                        *
10  *                                                                             *
11  *  This program is distributed in the hope that it will be useful,            *
12  *  but WITHOUT ANY WARRANTY; without even the implied warranty of             *
13  *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the              *
14  *  GNU General Public License for more details.                               *
15  *                                                                             *
16  *  You should have received a copy of the GNU General Public License          *
```

```
 *  along with this program. If not, see <http://www.gnu.org/licenses/>.    *
 *                                                                          *
 ****************************************************************************/

#include <atomic>
#include <iostream>
#include <linux/can.h>
#include <linux/can/raw.h>
#include <map>
#include <net/if.h>
#include <stdint.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <unistd.h>

#pragma once
```

2(b).1 Comments

```
/****************************************************************************
 *                                                                          *
 *  Copyright (C) 2025 RoboMaster.                                          *
 *  Illini RoboMaster @ University of Illinois at Urbana-Champaign          *
 *                                                                          *
 *  This program is free software: you can redistribute it and/or modify    *
 *  it under the terms of the GNU General Public License as published by    *
 *  the Free Software Foundation, either version 3 of the License, or       *
 *  (at your option) any later version.                                     *
 *                                                                          *
 *  This program is distributed in the hope that it will be useful,         *
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of          *
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the           *
 *  GNU General Public License for more details.                            *
 *                                                                          *
 *  You should have received a copy of the GNU General Public License       *
 *  along with this program. If not, see <http://www.gnu.org/licenses/>.    *
 *                                                                          *
 ****************************************************************************/
```

The file begins with a large comment block that contains *licensing* information. This is standard practice.

Remark | The most common open source licences you will see are MIT, Apache and GPL. MIT, Apache, and LGPL licenses are less restrictive, while most forms of the GPL are *copyleft* licenses. Copyleft licenses force all software that includes the licensed code to be copylefted and open source as well. Read more at https://www.gnu.org/licenses/license-list.html.

In C++, *comments* can be written in two ways:

- `/* comment */` for multi-line comments
- `// comment` for single-line comments

This should be self-explanatory.

Remark | Comments are not just for the user. They are often used to generate documentation by an IDE or external program (these are referred to as *docstrings*). Therefore you should try to follow a style guide when writing comments.

2(b).2 Include Statements

```
#include <atomic>
#include <iostream>
#include <linux/can.h>
#include <linux/can/raw.h>
#include <map>
#include <net/if.h>
#include <stdint.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <unistd.h>
```

The `#include` directive tells the compiler to include the contents of other files. You may notice there are two different types:

- `#include <filename>` for system headers (standard library and system libraries)
- `#include "filename"` for "local" headers (i.e. our own files)

Notice how we include both standard C++ libraries like `atomic` and `iostream`, as well as Linux system headers for CAN communication like `linux/can.h`.

Tip | You can run `g++ -H -fsyntax-only foo.h` in your terminal to find out where the headers included in `foo.h` are on your computer.

Remark | For me, `atomic` lives under `/usr/include/c++/15.2.1/atomic` while `linux/sys.h` lives at `/usr/include/linux/can.h`.

2(b).3 Pragma Once

```
#pragma once
```

The `#pragma once` directive is a *header guard* that tells the compiler to not include the same file multiple times. This is important because including the same file multiple times can lead to errors.

Remark | This is not magic; you can implement this manually using preprocessor directives (remember?) like `#ifndef`.

2(c) Constants and Namespaces

```
34 #define MAX_CAN_DEVICES 12
35
36 namespace CANRAW {
```

Here we define a constant `MAX_CAN_DEVICES` using the preprocessor directive `#define`. Constants defined this way in C++ are replaced by their values *before* compilation.

Here we meet a `namespace`. *Namespaces* are used to group related code and prevent naming conflicts. Everything within the `CANRAW` namespace can be accessed using the scope resolution operator `::`, i.e. `CANRAW::function_name`.

Remark | You have probably encountered namespaces in other programming languages. For example, variables in a function cannot be accesed outside of the function. C++ is unlike, for example, Python, in that it does this more explicitly.

2(d) Typedef and Function Pointers

```
37
38 typedef void (*can_rx_callback_t)(const uint8_t data[], void *args);
39
```

We meet `typedef`.

C++ is a *typed* language. If you do not know what a type or a type signature is, please consult a search engine. A `typedef` creates a new type from existing ones.

Remark | If you are familiar with Python, this is similar to a type alias. If I were to define an equivalent type in Python, it would look something like

```
from typing import *
CANRxCallbackType: TypeAlias = Callable[Tuple[Tuple[bytes], Any], None]
```

We `typedef` a pointer to such a function, hence

- `void`: The function returns nothing.
- `(*can_rx_callback_t)`: This is the name of our alias, with the asterisk indicating it is a pointer. We wrap it in parenthesis so it's a pointer to a function (`void (*func)`) and not a function that returns a void pointer (`(void *)func`).
- `(const uint8_t data[], void *args)`: These are the arguments the function takes. `const uint8_t data[]` is an array (`[]` after a variable name means an array) of constant (`const`, cannot be changed) unsigned 8-bit integers (`uint8_t`, `_t` indicates it is a type). The void pointer `void *args` means that it will accept a pointer to `args` of any type.

Remark | `void *` is *not* the null pointer type `std::nullptr_t`, which points to nothing. `void *` is a special pointer that can point to anything.

Therefore the type `can_rx_callback_t` indicates a pointer to a function which takes a constant array of bytes and a pointer to anything, and returns nothing.

2(e) Class Declaration

```
40 class CAN {
41 public:
42   CAN(const char *name = "can0");
43   ~CAN();
44   /**
45    * @brief Transmits a CAN message
46    * @param can_id The CAN ID to transmit to
47    * @param dat Pointer to the data to transmit
48    * @param len Length of the data in bytes
49    */
50   void Transmit(canid_t can_id, uint8_t *dat, int len);
51
52   /**
53    * @brief Receives a single CAN message
54    * @note This is a blocking call
55    */
56   void Receive();
57
58   /**
59    * @brief Closes the CAN socket and cleans up resources
60    */
61   void Close();
62
63   /**
64    * @brief Registers a callback for a specific CAN ID
65    * @param can_id The CAN ID to register for
66    * @param callback The callback function to invoke when message received
67    * @param args Optional arguments to pass to the callback
68    * @return 0 on success, -1 on failure
69    */
70   int RegisterCanDevice(canid_t can_id, can_rx_callback_t callback,
71                         void *args = nullptr);
72
73   /**
74    * @brief Deregisters a callback for a specific CAN ID
75    * @param can_id The CAN ID to deregister
76    * @return 0 on success, -1 if CAN ID not found
```

```
77      */
78     int DeregisterCanDevice(canid_t can_id);
79     struct can_frame frx;
80 
81     /**
82      * @brief Starts a thread to continuously receive CAN messages
83      * @param stop_flag Pointer to atomic bool to control thread execution
84      * @param interval_us Time between receive attempts in microseconds
85      */
86     std::atomic<bool> *StartReceiveThread(int interval_us = 10);
87     std::atomic<bool> *stop_flag_;
88 
89 private:
90     int s;
91     struct sockaddr_can addr;
92     struct ifreq ifr;
93     struct can_frame ftx;
94     std::map<canid_t, std::pair<can_rx_callback_t, void *>> callback_map;
95     std::atomic<bool> *receive_thread_present_;
96 };
```

This is the declaration of our main `CAN` class. Let's examine its components:

2(e).1 Access Specifiers

The class has two access specifiers, one on line 41 and the other on line 89:

- `public`: Members that can be accessed from outside the class
- `private`: Members that can only be accessed from within the class

2(e).2 Constructors and Destructors

```
42     CAN(const char *name = "can0");
43     ~CAN();
```

- The first line is the *constructor*. This is a special function with the same name as the class that gets called when we create an object of this class. The `name = "can0"` indicates the default value for `name` is "can0".
- The second line is the destructor. This is called when the object is destroyed and is used for cleanup.

REMARK | Note that `"can0"` is assigned to a (constant) `char *`. In C++, strings can be represented as an array of characters (a pointer is an array, and vice versa), or as a string from the standard library `std::string`.

2(e).3 Member Functions

The class declares several member functions:

- `Transmit` - For sending CAN messages.
- `Receive` - For receiving CAN messages.
- `Close` - For closing the CAN connection.
- `RegisterCanDevice` - For registering callbacks.
- `DeregisterCanDevice` - For removing callbacks.
- `StartReceiveThread` - For starting a background thread.

Let us take a look at `Transmit`:

```
50  /**
51   * @brief Transmits a CAN message
52   * @param can_id The CAN ID to transmit to
53   * @param dat Pointer to the data to transmit
54   * @param len Length of the data in bytes
55   */
56  void Transmit(canid_t can_id, uint8_t *dat, int len);
```

This function returns nothing (`void`) and takes three arguments.

- `canid`: `canid_t` comes from `linux/can.h`. It is just an alias for a 32-bit unsigned integer.
- `*dat`: A pointer to a byte array.
- `len`: The length of the byte array.

TIP | Notice the comment before the `Transmit` function declaration. Notice it is in some specific format. This is used to provide information to IDEs and automatic documentation generators. It is good practice to write these.

REMARK | Generally, it may not trivial to determine where an object ends in memory. In many methods, you may have to provide the length of a data structure yourself.

We will explain the member functions in further detail in section 3.

2(e).4 Member Variables

The class has several member variables:
Public variables:

- `struct can_frame frx;`, on line 79: The definition of `can_frame` can be found in `linux/can.h`. The prefix `struct` indicates that `can_frame` it is a struct. `frx` is short for "frame receive". We will be storing the CAN data we receive into this structure.
- `std::atomic<bool> *stop_flag_;`, on line 87: A pointer to an *atomic* boolean. You can think of an atomic variable as one that is thread safe. This flag stops the receive thread when set to true. I will elaborate on this in section 3.

Private variables:

- `int s;`, on line 90: An integer to store the socket file descriptor. I will elaborate on this in subsubsection 3(b).1.
- `struct sockaddr_can addr`, on line 91: `sockaddr_can` is a `struct` defined in `linux/can.h`. It is used in socket configuration. I will elaborate on this in subsubsection 3(b).6.
- `struct ifreq ifr;`, on line 92: `ifreq` is a `struct` defined in `net/if.h`. It is used in socket configuration. I will elaborate on this in subsubsection 3(b).5.
- `struct can_frame ftx;`, on line 93: This shares the same type as `frx`. `ftx` is short for "frame transmit"; we will be storing the data we want to send out into this structure.
- `std::map<canid_t, std::pair<can_rx_callback_t, void *>> callback_map;`, on line 94: Something to help us make callbacks. I will elaborate on this in subsubsection 3(e).5.
- `std::atomic<bool> *receive_thread_present_`, on line 95: Another atomic boolean, this one to make sure only one receive thread is running at a time.

2(f) Namespace Closing and a Note

```
97
98 } // namespace CANRAW
```

This closes the namespace we opened earlier. Notice the comment. If our file is littered with lots of `}`s, it is good to include these comments to remind ourselves what ends where.

Here is a comment from me. If there is anything basic you do not understand (or if you basically do not understand anything), take the time to search it up or ask an LLM. However, there is no need to fully understand the networking/operating system bits.

3 Understanding our CAN Source File

Now let us examine the implementation file `can.cc` to see how the functions declared in the header are actually implemented.

3(a) Include Statements and Namespace

```cpp
/*****************************************************************************
 *                                                                           *
 *  Copyright (C) 2025 RoboMaster.                                           *
 *  Illini RoboMaster @ University of Illinois at Urbana-Champaign           *
 *                                                                           *
 *  This program is free software: you can redistribute it and/or modify     *
 *  it under the terms of the GNU General Public License as published by     *
 *  the Free Software Foundation, either version 3 of the License, or        *
 *  (at your option) any later version.                                      *
 *                                                                           *
 *  This program is distributed in the hope that it will be useful,          *
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of           *
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the            *
 *  GNU General Public License for more details.                             *
 *                                                                           *
 *  You should have received a copy of the GNU General Public License        *
 *  along with this program. If not, see <http://www.gnu.org/licenses/>.     *
 *                                                                           *
 *****************************************************************************/
#include "can.h"
#include "utils.h"
#include <chrono>
#include <cstring>
#include <thread>

namespace CANRAW {
```

Notice how we include our own header file with quotes `"can.h"` rather than angle brackets (remember why?). `utils.h` contains basic functionality that I had to re-implement because of the outdated version of the `g++` the board uses.

- `chrono`: Provides tools to work with time. We use it for sleeping.
- `cstring`: Provides tools to work with strings in memory. This is the C++ version of C's `string.h` library.
- `thread`: Multithreading. You can think of this as running multiple pieces of code at once.

Remark | The `threading` library in Python exists but due to the GIL your program still runs on a single thread. You can search this up if you want.

We then re-enter the `CANRAW` namespace. If we did not, we would have to prefix everything we defined in can.h with `CANRAW::`. Exersise: why?

3(b) Constructor Implementation

```
28 CAN::CAN(const char *name) {
29   s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
30   if (s < 1) {
31     std::cerr << "Error while opening socket" << std::endl;
32   }
33
34   strcpy(ifr.ifr_name, name);
35   ioctl(s, SIOCGIFINDEX, &ifr);
36
37   addr.can_family = AF_CAN;
38   addr.can_ifindex = ifr.ifr_ifindex;
39
40   if (bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
41     std::cerr << "Error while binding address" << std::endl;
42   }
43
44   stop_flag_->store(false);
45   receive_thread_present_->store(false);
46 }
```

3(b).1 Scope Resolution

```
28 CAN::CAN(const char *name) {
```

The `CAN::` before the constructor name indicates that this function belongs to the `CAN` class within the `CANRAW` namespace. `const char *name` takes a single argument name.

Tip | Notice that we set a default value in can.h and did not repeat it here. This is fine.

3(b).2 Socket Creation

```
29 s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
```

This creates a socket.

Let us look at the declaration of this function in sys/socket.h:

```
99  /* Create a new socket of type TYPE in domain DOMAIN, using
100    protocol PROTOCOL.  If PROTOCOL is zero, one is chosen automatically.
101    Returns a file descriptor for the new socket, or -1 for errors.  */
102 extern int socket (int __domain, int __type, int __protocol) __THROW;
```

Now we know know what this means:

- `PF_CAN`: Protocol family for CAN.
- `SOCK_RAW`: Raw socket type.
- `CAN_RAW`: CAN raw protocol.

Remeber the type of `s`? It was an `int`. File descriptors, or fd for short, can be represented as integers.

REMARK | A socket is a file. In UNIX everything is represented as a file, including your keyboard, mouse, screen, etc. On a UNIX device, you can look under the 🗁`/dev` folder to see this.

3(b).3 Error Handling

```
30  if (s < 1) {
```

We check if the socket was created successfully by checking if `s < 1`. If there's an error, we print a message to `std::cerr` (standard error output). Do you know why? Hint: read the code block in subsubsection 3(b).2.

3(b).4 String Operations

```
34  strcpy(ifr.ifr_name, name);
```

This copies the interface name to the `ifr` structure. This is a C-style string operation. `strcpy` was provided by 🔖`cstring`.

3(b).5 System Calls

```
30  ioctl(s, SIOCGIFINDEX, &ifr);
```

This is a system call. `ioctl` is provided by 🔖`sys/ioctl.h`.

This performs the I/O control operation specified by REQUEST on FD. One argument may follow; its presence and type depend on REQUEST. (Taken directly from a comment in 🗎`/usr/include/sys/ioctl.h`.)

- `s`: The file descriptor of our CAN socket.
- `SIOCGIFINDEX`: Retrieve the interface index of the interface into `ifr_ifindex`. (Taken directly from ❯ `man netdevice`.)
- `&ifr`: A reference to a `struct ifreq` that will be populated with the interface information; specifically, the `SIOCGIFINDEX` request informs the kernel to fill in `ifr.ifr_ifindex` with the index of the network interface whose name is specified in `ifr.ifr_name`.

In human language, we are telling our computer to find the interface index of `s` and stick it into the `ifr_ifindex` field of a `ifreq` structure that we pass by reference.

3(b).6 Struct Initialization

```
addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;
```

After retrieving the interface index via `ioctl`, we initialize the `sockaddr_can` structure `addr`, which is used to bind the socket to a specific CAN interface.

- `addr.can_family = AF_CAN;`: Sets the address family to CAN. This tells the kernel we are working with CAN sockets.
- `addr.can_ifindex = ifr.ifr_ifindex;`: Assigns the interface index we obtained earlier from the `ifreq` structure. Populating a `struct` with configuration data before passing it to a system call is a common pattern in UNIX network programming. The `bind` function (called next) uses this fully initialized `addr` to associate our raw CAN socket with the desired hardware interface (e.g., `can0`).

Remark | In C and C++, structs are value types. When you declare a struct variable, its fields are uninitialized unless explicitly assigned.

3(b).7 Binding

```
if (bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
  std::cerr << "Error while binding address" << std::endl;
}
```

The `bind` system call associates the socket file descriptor `s` with a specific local address. In this case, the CAN interface we've configured in the `addr` structure.

Here is the docstring above `bind` in `sys/socket.h`:

```
/* Give the socket FD the local address ADDR (which is LEN bytes long).  */
extern int bind (int __fd, __CONST_SOCKADDR_ARG __addr, socklen_t __len)
     __THROW;
```

- `s`: The socket file descriptor returned by `socket()`.
- `(struct sockaddr *)&addr`: A pointer to the `sockaddr_can` structure we initialized earlier. We cast it to `sockaddr*` because `bind` is a generic function that works with any address family. It doesn't know about `sockaddr_can` specifically, so we must cast to its parent type.
- `sizeof(addr)`: The size of the address structure, so the kernel knows how much memory to read from the pointer.

If `bind` fails (returns `< 0`), we print an error to `std::cerr`, indicating the socket could not be bound to the specified interface.

Remark | In Linux networking, "binding" a socket means assigning it to a specific network interface and/or port. For CAN sockets, there's no "port"', instead, you bind to a physical or virtual CAN interface like `can0`.

After binding, our socket is now ready to send and receive CAN frames on the specified interface.

Remark | In POSIX systems, `stdout` (standard output) and `stderr` (standard error) are the two *file streams*. Both are typically output to the terminal by default, but they can be redirected independently. This separation is an arbitrary design choice made for practicality: it allows users to capture normal program output (`stdout`) while still seeing error messages (`stderr`) on screen, or vice versa. You can search up flow control operators for more information.

3(b).8 Atomic Operations

```
43   stop_flag_->store(false);
44   receive_thread_present_->store(false);
```

We use `->store` to set the atomic boolean values. Here we initialize the stop flag and the receive flag to `false`.

3(c) Destructor Implementation

```
48 CAN::~CAN() { this->Close(); }
```

The destructor just calls the `Close()` method. The `this->` syntax explicitly refers to the current object.

3(d) Transmit Method

```
50 void CAN::Transmit(canid_t can_id, uint8_t *dat, int len) {
51   ftx.can_id = can_id;
52   ftx.can_dlc = clip(len, 0, CAN_MAX_DLEN);
53   memcpy(ftx.data, dat, sizeof(uint8_t) * ftx.can_dlc);
54   if (write(s, &ftx, sizeof(struct can_frame)) != sizeof(struct can_frame)) {
55     std::cerr << "Error while sending CAN frame" << std::endl;
56   }
57 }
```

Let us go through `Transmit`.

3(d).1 Struct Field Assignment

```
51   ftx.can_id = can_id;
```

The first step in transmitting a CAN message is assigning the target `can_id` to the `can_id` field of the `ftx`. This field identifies which device or functional unit on the CAN bus should receive the message.

3(d).2 DLC and Utility Function

```
52   ftx.can_dlc = clip(len, 0, CAN_MAX_DLEN);
```

Here we assign the *Data Length Code* (`can_dlc`) using a helper function `clip`. Since CAN frames are limited to 8 bytes of data (defined by `CAN_MAX_DLEN`), this function ensures `len` is clamped within valid bounds to prevent buffer overruns and malformed frames. We use `clip` from our own `utils.h` because we are on a very old version of `g++` where `std::clamp` is not available.

```
template <typename T> T clip(T value, T min, T max) {
  return value < min ? min : (value > max ? max : value);
}
```

Read through `linux/can.h` and see if you can understand.

```
107 /**
108  * struct can_frame - Classical CAN frame structure (aka CAN 2.0B)
109  * @can_id:   CAN ID of the frame and CAN_*_FLAG flags, see canid_t definition
110  * @len:      CAN frame payload length in byte (0 .. 8)
111  * @can_dlc:  deprecated name for CAN frame payload length in byte (0 .. 8)
112  * @__pad:    padding
113  * @__res0:   reserved / padding
114  * @len8_dlc: optional DLC value (9 .. 15) at 8 byte payload length
115  *            len8_dlc contains values from 9 .. 15 when the payload length is
116  *            8 bytes but the DLC value (see ISO 11898-1) is greater then 8.
117  *            CAN_CTRLMODE_CC_LEN8_DLC flag has to be enabled in CAN driver.
118  * @data:     CAN frame payload (up to 8 byte)
119  */
120 struct can_frame {
121         canid_t can_id;  /* 32 bit CAN_ID + EFF/RTR/ERR flags */
122         union {
123                 /* CAN frame payload length in byte (0 .. CAN_MAX_DLEN)
124                  * was previously named can_dlc so we need to carry that
125                  * name for legacy support
126                  */
127                 __u8 len;
128                 __u8 can_dlc; /* deprecated */
129         } __attribute__((packed)); /* disable padding added in some ABIs */
130         __u8 __pad; /* padding */
131         __u8 __res0; /* reserved / padding */
132         __u8 len8_dlc; /* optional DLC for 8 byte payload length (9 .. 15) */
133         __u8 data[CAN_MAX_DLEN] __attribute__((aligned(8)));
134 };
```

Tip | We can see that `can_dlc` is marked deprecated. *Deprecated* means that a feature has been removed or will be removed soon. Do not depend on deprecated features.

3(d).3 Memory Copying

```
53    memcpy(ftx.data, dat, sizeof(uint8_t) * ftx.can_dlc);
```

We copy the user-provided data buffer `dat` into the `data` array of the `can_frame` structure. `memcpy` (from `cstring`) performs a low-level byte-by-byte copy which is efficient and safe for fixed-size buffers. Note that we use `ftx.can_dlc` (not `len`) to determine how many bytes to copy, ensuring we never exceed the legal payload size even if the caller passed a longer `len`.

3(d).4 Writing to Socket

```
54    if (write(s, &ftx, sizeof(struct can_frame)) != sizeof(struct can_frame)) {
55      std::cerr << "Error while sending CAN frame" << std::endl;
56    }
```

Finally we send the fully prepared `can_frame` through the socket file descriptor `s` using the POSIX `write` system call. The kernel interprets this as a request to transmit a raw CAN message over the bound interface. `write` returns the amount of bytes it wrote. If this is less than expected (or -1), it indicates an error.

Remark | Remember, everything can be thought of as a file. We send the message by writing our message to the file representing the CAN bus.

3(e) Receive Method

```
59  void CAN::Receive() {
60    if (read(s, &frx, sizeof(struct can_frame)) != sizeof(struct can_frame)) {
61      std::cerr << "Error while receiving CAN frame" << std::endl;
62      return;
63    }
64    auto it = callback_map.find(frx.can_id);
65    if (it != callback_map.end()) {
66      it->second.first(frx.data, it->second.second);
67    }
68  }
```

This method receives and distributes CAN frames.

3(e).1 Reading from Socket

```
60    if (read(s, &frx, sizeof(struct can_frame)) != sizeof(struct can_frame)) {
```

This reads a CAN frame from the socket. Note how this is like reading from a file. Again, these are the same. Can you guess what this does based on subsubsection 3(d).4?

3(e).2 Auto Keyword

```
60  auto it = callback_map.find(frx.can_id);
```

We use the `auto` keyword to automatically deduce the type of the iterator. This is a modern C++ feature that makes code more readable. `callback_map` is a map. I will elaborate on it here and the next few subsubsections. Recall its signature and the signature of `can_rx_callback_t`:

```
std::map<canid_t, std::pair<can_rx_callback_t, void *>> callback_map;
typedef void (*can_rx_callback_t)(const uint8_t data[], void *args);
```

Using `auto` for the iterator type avoids having to write out the full type:

```
std::map<canid_t, std::pair<can_rx_callback_t, void *>>::iterator it =
  callback_map.find(frx.can_id);
```

which is a mess.

3(e).3 Map Lookup

We associate a CAN identifier (`canid_t`) with a callback function and an optional context pointer (`void *`). When a CAN message is received, the system looks up the corresponding `can_id` in `callback_map` with the find `find` method for maps (which returns an iterator).

3(e).4 Iterator Usage

We check if the iterator is valid (`it != callback_map.end()`) and then call the callback function with `it->second.first(frx.data, it->second.second)`.

3(e).5 Callback Function

Recall that `callback_map` is a `std::map` with the following type signature:

```
std::map<canid_t, std::pair<can_rx_callback_t, void *>> callback_map;
```

This means each element in the map is a key-value pair where:

- The **key** is of type `canid_t` (the CAN identifier).
- The **value** is of type `std::pair<can_rx_callback_t, void *>`, which itself contains two elements:
  - `first`: A function pointer of type `can_rx_callback_t`. This is the callback function to invoke.
  - `second`: A `void *`. This is an optional user-provided argument (context) to pass to the callback.

When we call `callback_map.find(frx.can_id)`, we get an iterator `it` pointing to the found entry (or `callback_map.end()` if not found).

Assuming the lookup succeeds (`it != callback_map.end()`), then:

- `it->second` accesses `std::pair<can_rx_callback_t, void *>`, the value part of the kv pair.
- `it->second.first` accesses the first element of that pair, the pointer to the function to be called.
- `it->second.second` accesses the second element of that pair, the pointer to the context (`void *`) to be passed to the callback.

Therefore, the full expression:

```
it->second.first(frx.data, it->second.second);
```

In English, this means: "If we have a callback registered for this CAN ID, call that function now and give it the received data along with any user-specified context."

This mechanism allows different parts of the system to register interest in specific CAN IDs and respond automatically when messages arrive.

Remark | This pattern is common in embedded systems and robotics: decoupling message reception from processing logic via callbacks. It promotes clean architecture and scalability so you can add new handlers for new CAN IDs/types without modifying existing code.

3(f) Device Registration Methods

```
70 int CAN::RegisterCanDevice(canid_t can_id, can_rx_callback_t callback,
71                            void *args) {
72   if (callback_map.size() >= MAX_CAN_DEVICES) {
73     std::cerr << "Maximum number of CAN devices reached" << std::endl;
74     return -1;
75   }
76   callback_map[can_id] = std::make_pair(callback, args);
77   return 0;
78 }
79
80 int CAN::DeregisterCanDevice(canid_t can_id) {
81   auto it = callback_map.find(can_id);
82   if (it != callback_map.end()) {
83     callback_map.erase(it);
84     return 0;
85   }
86   std::cerr << "Can ID " << can_id << " not registered" << std::endl;
87   return -1;
88 }
```

These methods are for managing the callback map.

3(f).1 Size Checking

```
72    if (callback_map.size() >= MAX_CAN_DEVICES) {
73      std::cerr << "Maximum number of CAN devices reached" << std::endl;
74      return -1;
75    }
```

We check if we've reached the maximum number of devices before adding a new one.

3(f).2 Pair Creation

```
76    callback_map[can_id] = std::make_pair(callback, args);
```

This creates a pair of values to store in our map.

3(f).3 Map Erasure

```
76    auto it = callback_map.find(can_id);
77    if (it != callback_map.end()) {
78      callback_map.erase(it);
79      return 0;
80    }
```

This attempts to find and remove an entry from the map.

3(g) Thread Management

```
90  std::atomic<bool> *CAN::StartReceiveThread(int interval_us) {
91    if (receive_thread_present_->load()) {
92      std::cerr << "Error: Receive thread already running" << std::endl;
93      return nullptr;
94    }
95
96    stop_flag_->store(false);
97    receive_thread_present_->store(true);
98    std::thread([this, interval_us]() {
99      while (!stop_flag_->load()) {
100       this->Receive();
101       std::this_thread::sleep_for(std::chrono::microseconds(interval_us));
102     }
103     receive_thread_present_->store(false);
104   }).detach();
105
106   return stop_flag_;
107 }
```

This method starts a background thread for receiving CAN messages:

3(g).1 Lambda Functions

```
98   std::thread([this, interval_us]()
```

The `[this, interval_us]() { ... }` syntax creates a lambda function (anonymous function). The variables in the square brackets (`this` and `interval_us`) mean that those variables are available inside the scope of the lambda function.

Remark | Lambda functions allow us to define functions wherever we need with less cumbersome syntax. It is often used when passing single-use functions to another function, i.e. a sort or filter function.

3(g).2 Thread

```
98   std::thread([this, interval_us]() {
99     while (!stop_flag_->load()) {
100      this->Receive();
101      std::this_thread::sleep_for(std::chrono::microseconds(interval_us));
102    }
103    receive_thread_present_->store(false);
104  }).detach();
```

We first call the `Receive` function, then we call `sleep_for`. This pauses the thread for a specified amount of time. Notice our while loop checks for `!stop_flag_->load()`. When the stop flag is set, the thread stops.

3(g).3 Thread Detachment

`.detach()` detaches the thread so it runs independently.

3(h) Close Method

```
109 void CAN::Close() {
110   // Signal receive thread to stop
111   stop_flag_->store(true);
112   receive_thread_present_->store(false);
113   // Close socket
114   close(s);
115 }
```

This method cleans up resources by stopping the receive thread and closing the socket.

4 Using the CAN Class: Examples

Let's look at how to use the CAN class in practice by examining the example files.

4(a) CAN Receive Example

```
21 #include "board/can.h"
22 #include "stdint.h"
23 #include "stdio.h"
24 #include <unistd.h>
25
26 void receive(CANRAW::CAN *can0) {
27   // CANRAW::CAN *can0 = new CANRAW::CAN("can0");
28   // read(, &can0->frx, sizeof(struct can_frame));
29   can0->Receive();
30   for (int i = 0; i < 8; i++) {
31     printf("%02x ", (int)can0->frx.data[i]);
32   }
33   printf("\n");
34 }
35
36 int main() {
37   CANRAW::CAN *can0 = new CANRAW::CAN("can0");
38   printf("Start can receive test\n");
39   while (true) {
40     receive(can0);
41   }
42   return 0;
43 }
```

This is src/examples/can_receive.cc.

4(a).1 Object Creation

```
37   CANRAW::CAN *can0 = new CANRAW::CAN("can0");
```

This creates a new CAN object on the heap using the `new` operator.

4(a).2 Method Calls

```
29   can0->Receive();
```

This calls the `Receive` method using the arrow operator (`->`) to access methods on a pointer.

4(a).3 Infinite Loop

```
39   while (true) {
```

This creates an infinite loop. We do this when there is something that must be run continuously.

4(b) CAN Send Example

```
21 #include "board/can.h"
22
23 void sendtest() {
24   int i;
25   int len = 8;
26   uint8_t dat[8] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00};
27
28   CANRAW::CAN *can0 = new CANRAW::CAN("can0");
29
30   for (i = 0; i < 10; i++) {
31     can0->Transmit(0x200, dat, len);
32   }
33   can0->Close();
34 }
35
36 int main() {
37   std::cout << "Start can send test" << std::endl;
38   sendtest();
39   std::cout << "End can send test" << std::endl;
40
41   return 0;
42 }
```

This is src/examples/can_send.cc. Try to figure this one out for yourself.

4(c) Receive Thread Example

```
21 #include "board/can.h"
22 #include "motor/motor.h"
23 #include <unistd.h>
24
25 int main() {
26   CANRAW::CAN *can = new CANRAW::CAN("can0");
27   control::MotorCANBase *motor = new control::Motor3508(can, 0x207);
28   control::MotorCANBase *motors[] = {motor};
29
30   std::atomic<bool> *can_stop = can->StartReceiveThread();
```

```
31    if (can_stop == nullptr) {
32      std::cerr << "Error: Could not start CAN receive thread" << std::endl;
33      return 1;
34    }
35
36    while (true) {
37      motor->SetOutput(400);
38      control::MotorCANBase::TransmitOutput(motors, 1);
39      motor->PrintData();
40      usleep(100);
41    }
42
43    return 0;
44  }
```

This is src/examples/motor_m3508.cc. Take note of lines 26, 27 and 30. Can you figure out what they do?

5 Concepts Covered

We breifly went over the following:

5(a) Basic Syntax

- Comments (`/* */` and `//`)
- Include statements (`#include`)
- Preprocessor directives (`#define`, `#pragma once`)
- Header and implementation files

5(b) Object-Oriented Programming

- Classes and objects
- Access specifiers (`public`, `private`)
- Constructors and destructors
- Member functions and variables
- Namespaces

5(c) Memory Management

- Pointers and the `*` and `->` operators
- Destructors

5(d) Standard Library

- Containers (`std::map`)
- Utility classes (`std::pair`)
- Thread support (`std::thread`, `std::atomic`)
- Time functions (`std::chrono`)

5(e) System Programming

- Linux socket API
- System calls (`socket`, `bind`, `ioctl`)
- File descriptor operations (`read`, `write`, `close`)

5(f) Modern C++ Features

- The `auto` keyword
- Lambda functions

6 Conclusion

This is the first tutorial, going over C++ and the Linux usage of CAN. There will be a shorter second part, introducing the actual structure of a CAN packet and how we structure data, i.e. communicating with multiple devices with a single packet.

If there are any concepts I have failed to cover, or you do not completely understand, search it up and ask an LLM.