

Designentscheidungen

Zweck des Dokuments

Dies ist eine Erläuterung der für das Kommandozeilenprogramm QtRoboParser getroffenen Designentscheidungen.

Zielpattform

Das Programm ist hauptsächlich für Unix basierte Systeme verwendbar, da es die Events von QtRobo auf einem erstellten Unix Domain Socket entgegennimmt. Unix Domain Sockets sind seit dem Insider Build 17063 zwar auch unter Windows mit einer laut Microsoft möglichst kompatiblen Implementierung verfügbar, allerdings wurde QtRoboParser nur unter Arch Linux getestet, wodurch keine Informationen zu notwendigen Anpassungen für eine Portierung bestehen.

Interaktion mit dem Nutzenden

Vorgabe für QtRoboParser ist es, möglichst flexibel zu sein. Deshalb sind die für Events verwendeten Präfixe und der Pfad zum verwendeten Socket konfigurierbar. QtRoboParser besitzt keine grafische Benutzeroberfläche, da die Verwendung im Terminal vorgesehen ist, um die SumD Event-Frames an andere Prozesse per Pipe weiterleiten zu können. Die Konfiguration muss deshalb über das Kommandozeileninterface erfolgen.

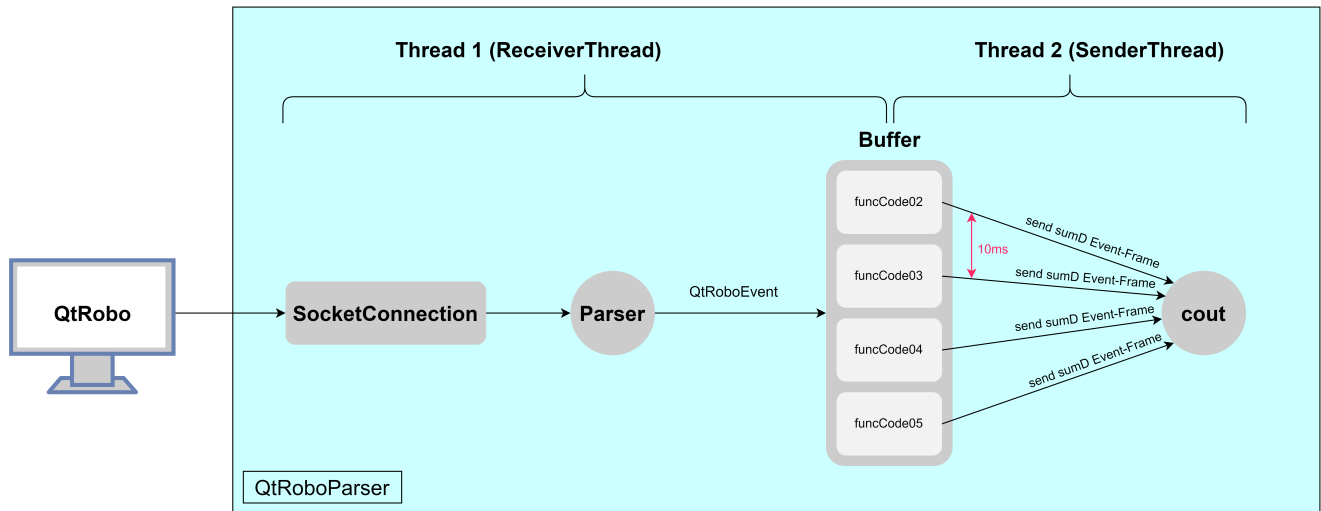
Programmiertechnik

QtRoboParser ist in C++ geschrieben. Es wurde darauf geachtet C++ Datenstrukturen zu verwenden. Das bedeutet, dass zum Beispiel `std::array` und `std::vector` anstatt von rohen C Arrays eingesetzt werden. Des Weiteren wurde darauf geachtet, Parameter wenn möglich als konstante Referenzen (`const&`) zu übergeben, alle Objekte uniform mit geschweiften Klammern anstatt runden Klammern zu initialisieren sowie Konstanten als konstante Ausdrücke zu definieren (`constexpr`).

Das fördert die Verständlichkeit und Lesbarkeit des Quellcodes. C-typische Fehler wie Zugriffe auf Speicheradressen außerhalb des definierten Arrays werden durch diese Designentscheidungen vermieden.

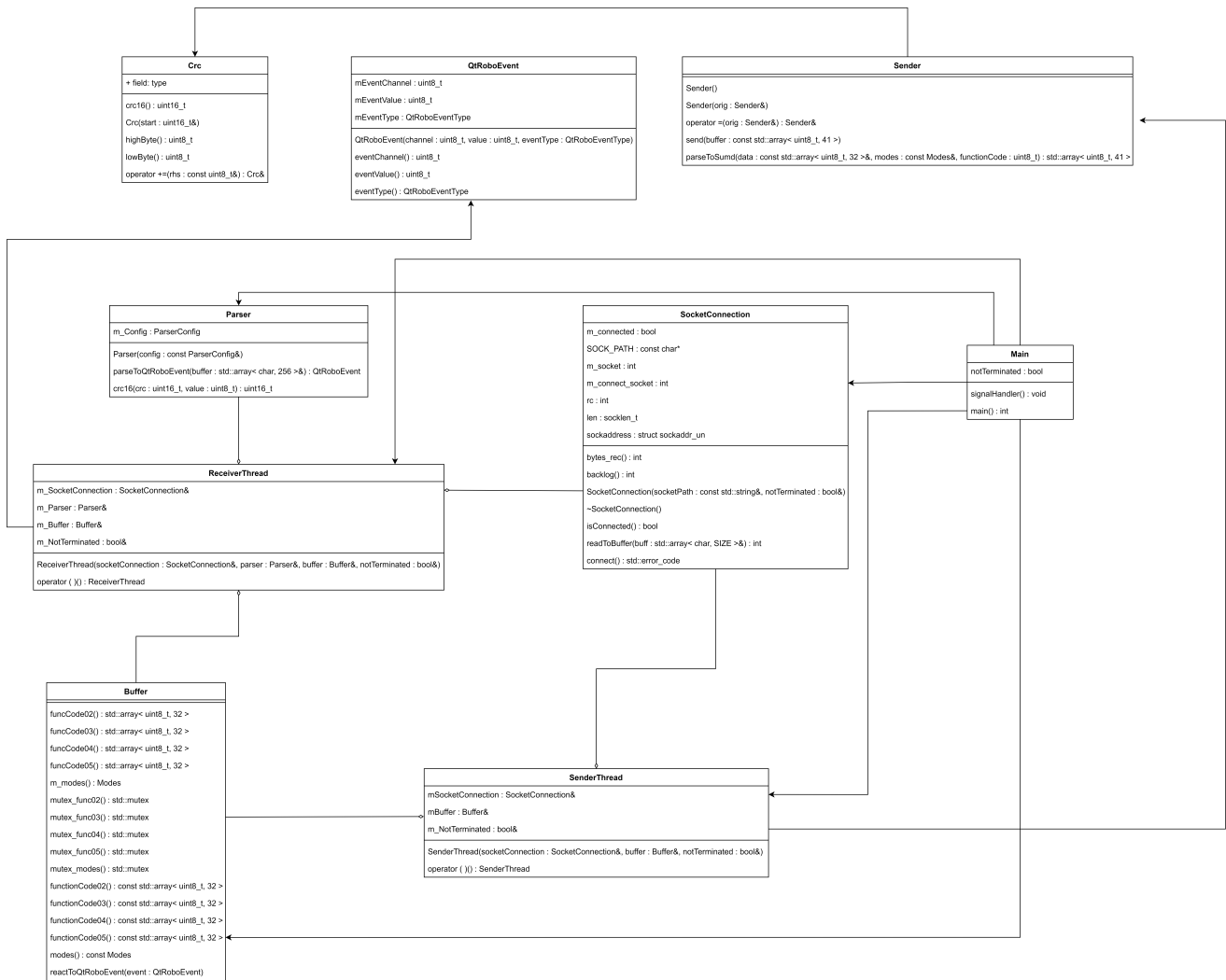
Funktionsweise

Die Anwendung erstellt einen Socket an dem per Kommandozeilenparameter übergebenen Pfad und wartet auf eine Verbindung von einer Gegenstelle. Die dort eingehenden Zeilen werden einzeln in QtRoboEvents geparkt, welche die standardmäßig auf 0 für proportionale bzw. false für binäre Controls gesetzten Werte verändern. Dieser Buffer muss in mehreren Teilen (function Codes) verschickt werden. Diese sind im SumD Protokoll festgelegt. Die folgende Abbildung zeigt die beschriebene Funktion:



Klassenlayout und Datentypen

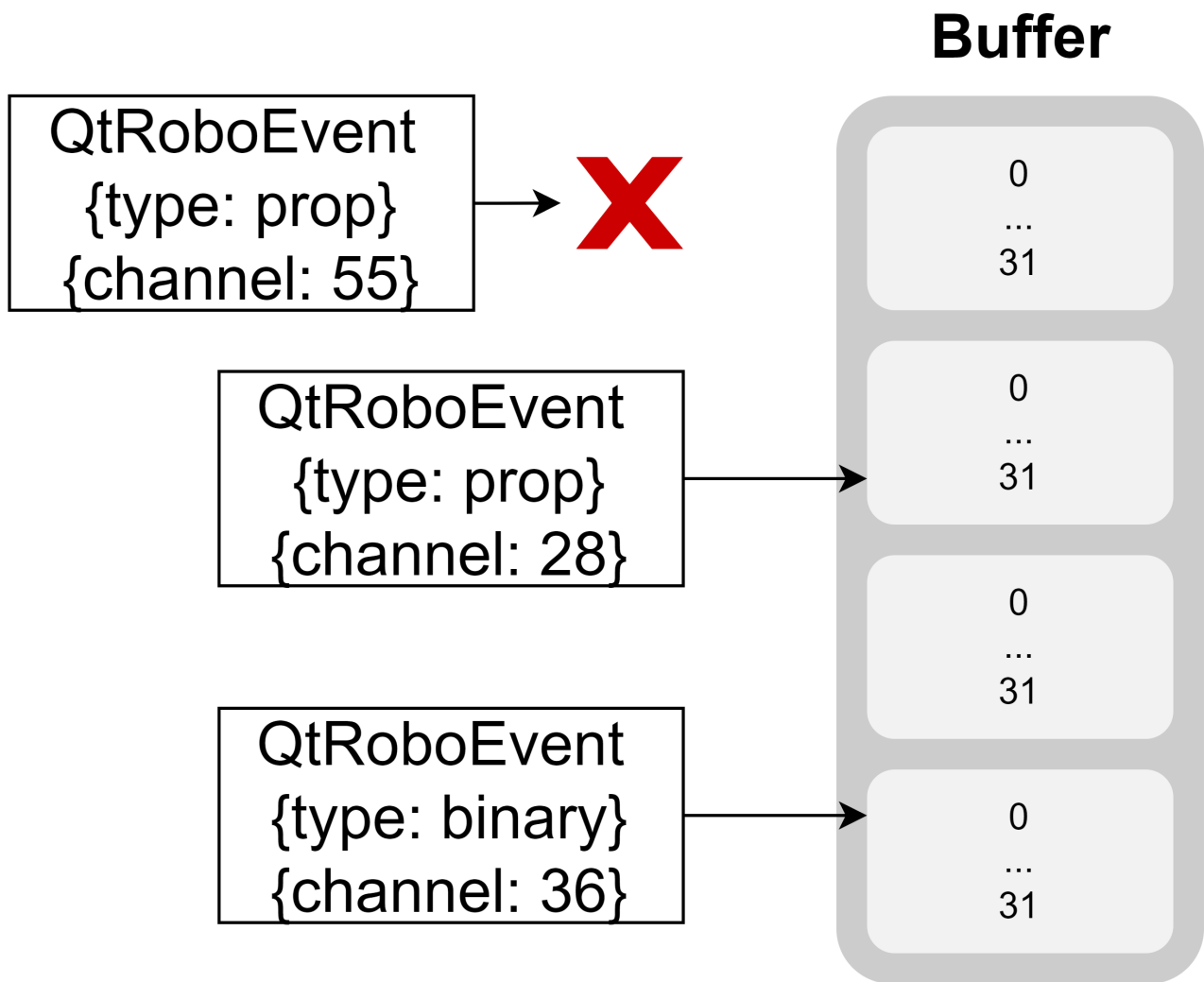
Dementsprechend gibt es mehrere Möglichkeiten die Applikation in Klassen zu unterteilen. Um der Anwendungsdomäne zu entsprechen, wurde sich für eine SocketConnection Klasse, eine Parser Klasse und eine Sender Klasse entschieden. Als primitive Datentypen für Ganzzahlwerte werden wenn immer möglich unsigned Typen verwendet (uint8_t, uint16_t etc.). Das ist möglich, da QtRobo nur positive ganze Zahlen bis maximal 100 versendet und die unterstützten Kanäle des SumD Protokolls nicht über 96 hinaus gehen. Dadurch werden Fehler durch negative Zahlen und die Umwandlung zwischen signed und unsigned Datentypen vermieden.



Buffer

Parser und Sender operieren auf der Datenstruktur Buffer. Der Buffer ist bereits in die als Function Codes definierten Abschnitte unterteilt. Der für das Senden zuständige Thread muss aus diesem Grund keine Rechenzeit für das Zerteilen des gesamten Buffers in die definierten Abschnitte aufwenden. Das geschieht bereits beim Auswerten eines QtRoboEvents. Ein QtRoboEvent bezieht sich auf einen bestimmten Index im Buffer, die Einsortierung ist daher einfach. Der in QtRobo angegebene Channel bezieht sich auf ein Element in einem der vier Arrays (Abschnitte des Buffers).

Die SocketConnection Klasse erzeugt zu Beginn den Socket und akzeptiert eingehende Verbindungen. Wird eine Verbindung beendet, löscht sie den Buffer und wartet auf eine erneute Verbindung eines Clients.



Domänenspezifische Datenstrukturen

Das sumD Protokoll erfordert die Berechnung einer Prüfsumme mit dem CRC16 Verfahren. Dafür werden alle Datenwerte und der Header des Event-Frames nacheinander zu einer Zahl im Datentyp `uint16_t` verrechnet. Anstatt für diese Berechnung eine entsprechende Funktion aus einer Klasse aufzurufen, wird ein domänenspezifischer Datentyp verwendet. Dieser hat den `uint16_t` Wert als Membervariable und den überladenen `+` Operator zum verrechnen eines neuen Wertes. Zusätzlich besitzt die Klasse getter um die Bytes der `uint16_t` Variable einzeln zu verwenden. Das ist hilfreich, da das sumD Protokoll byte-basiert ist.

Threading

Laut SumD Protokoll muss alle 10ms ein Event-Frame mit einem Teil des Buffers, abhängig vom Function Code versendet werden. Diese Funktionalität muss in einen separat laufenden Thread

ausgelagert werden, damit das Senden nicht durch eintreffende QtRoboEvents blockiert wird. Parser und Sender laufen daher in getrennten Threads und synchronisieren sich auf einen Buffer. Hier hat der Sender Priorität, da es besser ist einen veralteten Wert zu senden, als die Spezifikation des SumD Protokolls zu brechen und gegebenenfalls Verbindungsprobleme zu verursachen.

Damit auf den gleichen Daten operiert wird, werden die Verbindung zum Socket, der Buffer, sowie der Indikator für die Beendigung der Anwendung durch den Nutzenden per Referenz übergeben.

Signal Handling

Der ReceiverThread, welcher die Daten zeilenweise von der SocketConnection erhält und über den Parser in den Buffer überträgt, hat mit der Funktion `readToBuffer()` aus der SocketConnection einen blockierenden IO-Aufruf. Wird die Anwendung durch das Signal SIGINT vom Nutzenden beendet, bekommt der Main Thread das mit und setzt die Variable zum Abbrechen im ReceiverThread. Steht dieser gerade im blockierenden Aufruf, wird die Anwendung trotz Signal nicht beendet. Im Main Thread würde der blockierende Aufruf mit dem richtigen Signal Handler beendet werden. Andere Threads im gleichen Prozess bekommen von diesem Signal jedoch nichts mit.

Um das Programm auch in diesem Fall kontrolliert zu beenden, sind die Sockets nicht blockierend. Das wird mit dem Flag `O_NONBLOCKING` erzielt. Damit kehrt der IO Aufruf direkt zurück. Im Fall das nichts gelesen werden konnte, wird `errno` mit dem Pseudo-Error Wert `EAGAIN` belegt. Dementsprechend findet entweder die Verarbeitung der Daten statt, oder nicht.

Parser

Der Parser nimmt sich jeweils eine Zeile aus dem Socket und parst deren Kanal und ggf. deren Wert in ein QtRoboEvent. Zu Beginn wurde dieser Parser als Zustandsautomat umgesetzt. Dies bietet sich an, solange Eventtypen nur durch ihre Syntax unterscheidbar sind. Mit der Einführung der Präfixe pro Eventtyp ist diese Unterscheidung anhand der ersten Buchstaben möglich. Je nach Eventtyp wird mit einem regulären Ausdruck der Kanal und der neue Wert geparst. Hat der Eventtyp weder Kanal noch Wert, handelt es sich um den `MODE_CMD` bzw. den `SUB_CMD` Eventtyp. In diesem Fall wird nur der EventTyp im QtRoboEvent hinterlegt.

Fehlerbehandlung

In Linux entspricht jede Ressource einer Datei. Die Verbindung zum Unix Domain Socket zählt auch dazu. Der Zugriff darauf wird dementsprechend mit IO Funktionen realisiert. Diese können aus vielen Gründen fehlschlagen (siehe `errno` Werte). Um Seiteneffekte in der SocketConnection Klasse zu vermeiden, gibt die Funktion `connect()` einen als Enum Wert definierten Fehlerwert zurück. So kann der jeweilige Aufrufer der Klasse entscheiden wie mit dem Fehler umgegangen werden soll. Einziger Nachteil dieser Entscheidung ist die fehlende Kapselung von allen SocketConnection Aufrufen in einer Klasse, weil die Fehlerbehandlung beim Aufrufer stattfindet.

In Main wird die Anwendung bei einem fatalen Problem während der Erstellung des Sockets beendet. Gibt es allerdings ein eventuell clientseitiges Problem, wartet QtRoboParser erneut auf die

Herstellung einer Verbindung.