

# The Robotics and Mechatronics Kit “qfix”

Stefan Enderle

Neural Information Processing Department  
University of Ulm, Germany  
enderle@neuro.informatik.uni-ulm.de

**Abstract.** Robot building projects are increasingly used in schools and universities to raise the interest of students in technical subjects. They can especially be used to teach the three mechatronics areas at the same time: mechanics, electronics, and software. However, it is hard to find reusable, robust, modular and cost-effective robot development kits in the market. Here, we present *qfix*, a modular construction kit for edutainment robotics and mechatronics experiments which fulfills all of the above requirements and receives strong interest from schools and universities. The outstanding advantages of this kit family are the solid aluminium elements, the modular controller boards, and the programming tools which reach from an easy-to-use graphical programming environment to a powerful C++ library for the GNU compiler collection.

## 1 Introduction

Robot building projects are a good means to bring the interesting field of robotics to schools, high-schools, and universities. Studying robotics the students learn a lot about mechanics, electronics, and software engineering. Additionally, they can be highly motivated and learn to work in a team.

Performing a lot of robot building labs with pupils and students, we found that there is a gap between the relatively cheap toy-like kits, like LEGO Mindstorms or Fischertechnik Robotics and the quite expensive off-the-shelf robots. The toy kits offer a good opportunity to start building robots, but they mostly support the control of only 2 or 3 motors and the same number of sensors. Off-the-shelf robots (see e.g. [1,4,7]) are completely built up, so typically only the programming of the robot can be studied.

Alternatively, there exist a number of controllers, like the 6.270 board or the HandyBoard [3] which come without mechanical parts and so must be used in combination with other toy kits, like RC-controlled cars, or custom-built robots. However, these boards, can control only small motors and are not very expandable.

After building RoboCup robots from scratch [6,9,2,11] and supporting schools developing their own RoboCupJunior robot [10], the authors gained a lot of experience about reasonable mechanical concepts and controller architectures for a usable robot development kit. Thus, we decided to develop the robot kit family *qfix* and to provide it to schools and universities.

## 2 The *qfix* Approach: Modularity

The main concept behind *qfix* is modularity in the following dimensions:

- **Mechanics:** The mechanical parts are aluminium parts including rods, plates and holders for different sensors and actuators. These parts are the building blocks for constructing mechanical and mechatronic systems, like cars, walking robots, etc. Most parts contain threads and can easily be screwed together, so very robust models can be build.
- **Electromechanics and Electronics:** There already exist many compatible electromechanical and electrical parts including a variety of sensors, actuators, and controller boards. With these components it is possible to make the mechanical models *move* (by DC motors, servo-motors, stepper motors), *sense* (by tactile, infrared and ultrasonic sensors), and *think* (by powerful controller boards which can be programmed on the PC).
- **Software:** In the software area, modularity is no big deal. The *qfix* software comes with the powerful free GNU C++ toolchain (WinAVR for windows, respective libraries or RPMs for linux). Additionally, it contains an easy-to-use C++ class library for accessing all *qfix* electronics components.

Since beginners, say, of an age from 12, have problems going directly into C or C++ programming, we developed a graphical programming environment called GRAPE in order to simplify the programming of self-built robots. This software directly produces C++ code from the graphical description and thus supports the beginner in learning object-oriented programming.

### 2.1 Mechanics

The basic building blocks of the *qfix* system are anodized aluminium rods with  $\phi 6$  holes along all four sides and two M6 threads on the front and back side (see Fig. 1). Currently, there are rods from 20mm to 100mm including  $45^\circ$  rods.

Other basic elements are a variety of plates with holes and threads. These plates can be bolted to the rods using a screw and a nut or only a screw exploiting the rods’ frontal threads. Like the rods, the plates are given in different lengths and widths, currently up to 200mm x 200mm (see Fig. 2 for an exemplary plate).

All mechanical parts use holes and threads according to DIN/ISO standards and have a grid of 10mm.

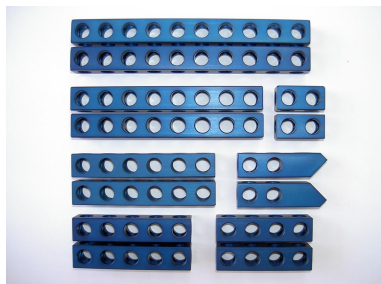
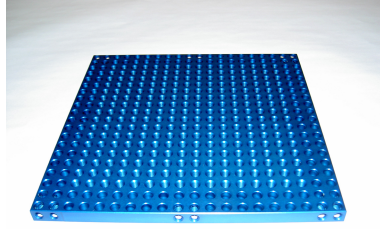


Fig. 1. Basic elements: rods



**Fig. 2.** Basic elements: plates (Here: 200x200mm with 400 threads)



**Fig. 3.** Wheels, axes, and gears

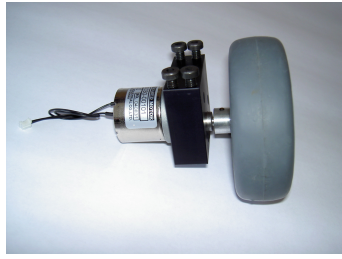
In Figure 3, some additional mechanical elements can be seen: wheels, casters, gears, and axes. They are usually used to implement dynamic models which then can be driven by different motors as shown in the next section.

## 2.2 Electromechanics/Electronics

**Motors:** In order to make a model move, motors are needed. Typical robotics applications often use different kinds of motors for different tasks: DC motors, servo motors, and stepper motors. *qfix* supports these different categories by providing the respective motor bearings (see Fig. 4) and electronics components for driving motor and wheel encoders.

**Sensors:** When building robots, it is also necessary to make them able to gather information about their environment. This can be done by mounting simple switches signalling bumps into obstacles, or by adding distance measuring devices like infrared or sonar sensors. As with motors, *qfix* supports numerous sensors by providing the respective bearing (see Fig. 5) for mounting the sensor to the model.

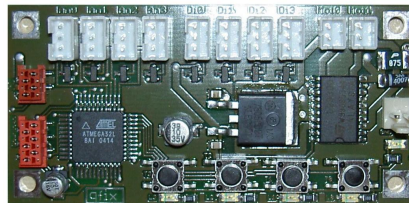
**Controllers:** Obviously, the motors and sensors must be driven by an electronics component. For *qfix*, we developed a new, modular controller board architecture which is both powerful and easy-to-handle. The board with the smallest controller is the “BobbyBoard” (see Figure 6) which uses the Atmel ATmega32 controller and supports the following I/Os:



**Fig. 4.** Exemplary motor bearing for a DC motor



**Fig. 5.** Exemplary sensor bearing for an IR distance sensor



**Fig. 6.** “BobbyBoard”: controller board with ATmega32

- 2 DC motor controllers (battery voltage, 1A)
- 4 digital inputs (0/5V)
- 4 analog inputs (0-5V)
- 8 digital outputs (battery voltage, 100mA)
- 4 LEDs
- 4 buttons
- I<sup>2</sup>C-bus for extensions

Further existing main boards are the “CAN128Board” which shows the same I/O capabilities but uses an Atmel AT90CAN128 controller with CAN interface, more memory and more speed. And, the “SoccerBoard” with 8 analog and 8 digital inputs, 8 digital outputs, 6 motor drivers, and optional CAN and USB interface (see Figure 7). This board is specifically designed for the requirements

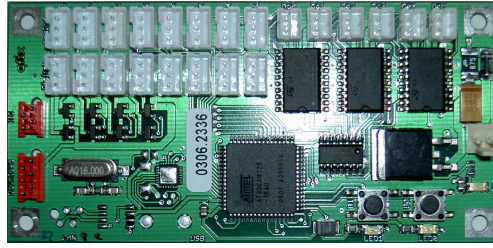


Fig. 7. “SoccerBoard”: controller board with ATmega128



Fig. 8. “LC-display board”: expansion board with LC-display

of RoboCupJunior, where often omnidrive platforms with three driven wheels plus a kicker and a “dribbler” are used combined with multiple sensor systems.

All controller boards are programmed (“flashed”) from the PC via a serial, parallel or USB link and then run autonomously without the host computer.

**Extension Boards:** The main idea behind the *qfix* boards is their flexible modular architecture: The main controller board runs the main program and communicates with expansion boards for setting actuator values and getting sensor data. The expansion boards themselves are responsible for controlling the attached devices, so the main processor does not have to perform expensive tasks, like feedback motor control, etc.

The controller boards contain an I<sup>2</sup>C-bus and optionally a CAN bus which both allow to chain dozens of boards of the same or different kinds to a large controller network. So, it is possible to either control more I/Os or even to implement distributed applications with decentralized control (see e.g. [5]).

The following extension boards based on I<sup>2</sup>C-bus are currently available:

- **Servo board 1:** The servo-board uses a Atmel mega8 for controlling 4 servo motors independently.
- **Servo board 2:** This servo-board is designed for humanoid robots and can control 24 servo motors independently. It contains a mega128 controller and a Xilinx FPGA for fast I/O control.
- **Stepper-board:** The stepper-board can control 4 stepper motors independently. Both, full and half step mode are supported.

- **DC-power board:** The DC-power-board is capable of controlling two DC motors with 4A each. It also contains two encoder input lines for each motor.
- **LC-display board:** An LC-display with 4 lines of 20 characters each (see Figure 8).
- **Relais boards:** There are two relais boards: one to be connected to the digital output of the controller board and one to be connected via the I<sup>2</sup>C-bus.

Further expansion boards, e.g. for Polaroid sonar sensors [8] and a camera board are currently under development.

## 2.3 Software

With *qfix* we provide the free GNU C++ toolchain including generic tools for downloading programs to the controller boards. Additionally, we provide a C++ class library supporting all *qfix* boards. On Windows, the generic tools mainly consist of the WinAVR GCC environment for Atmel controllers which includes the extensible editor *programmers notepad* and powerful download tools, like *avrdude*. All tools also run on Linux/Unix and Mac, so cross-platform development is fully supported.

The easy-to-use *qfix* C++ class library hides the low-level hardware interface from the programmer and supports the complete *qfix* extension board family. The main idea is to provide a specific C++ class for each *qfix* module. Therefore, the library provides the classes `BobbyBoard`, `SoccerBoard`, `LCD`, `SlaveBoard`, `StepperBoard`, `ServoBoard`, `RelaisBoard`, etc. For example, when building an application with the `BobbyBoard` and the `LCD` you use the respective classes, like the following:

```
#include "qfixBobbyBoard.h"    // include BobbyBoard library
#include "qfixLCD.h"           // include LCD library

int main()
{
    BobbyBoard board;           // construct object "board"
    LCD lcd;                   // construct object "lcd"

    board.ledOn(0);            // turn on LED 0
    board.waitForButton(0);    // wait until button 0 is pressed
    board.motor(0,255);        // turn on motor 0 to full speed
    lcd.print("Engines running"); // print a text on the LCD
}
```

As can be seen from the comments of the code, two instances of two classes are constructed: `board` and `lcd`. Their methods are called in order to let the main board turn on a LED and a motor, wait for a button press, and output text on the LCD.

A lot of the functionality is hidden in the constructors of both classes. When constructing the object `board` for instance, the constructor initializes all I/O

pins and starts an interrupt routine for motor PWM control. When constructing `lcd`, the constructor opens an I<sup>2</sup>C-bus channel and starts communicating with the physically connected LC-display. This mechanism works perfectly as long as expansion boards of different types are used only.

When using multiple expansion boards of the same type, the extended construction syntax can be used in order to connect the objects to the correct physical boards. Imagine you have a controller board and three identical LC-display boards:

```
#include "qfixBobbyBoard.h"    // include BobbyBoard library
#include "qfixLCD.h"           // include LCD library

int main()
{
    BobbyBoard board;          // construct object "board"
    LCD        lcd0(0);        // construct object "lcd0"
    LCD        lcd1(1);        // construct object "lcd1"
    LCD        lcd2(2);        // construct object "lcd2"

    board.waitForButton(0);    // wait until button 0 is pressed
    lcd0.print("Hallo");       // print a text on LCD 0
    lcd1.print("World!");      // print a text on LCD 1
    lcd2.print("Engines running"); // print a text on LCD 2
}
```

In this example, each of the three `lcdX` objects is connected to the physical LCD board with the respective ID. This ID can be hardcoded to the LCD by flashing the LCD board, or it can be dynamically changed by calling the method `lcd.changeID(newID)`.

For those who want to connect multiple controller boards but do not want to go into detail with programming the I<sup>2</sup>C-bus, we provide a class `SlaveBoard` which can be used as a “remote control” for connected BobbyBoard main boards:

```
#include "qfixBobbyBoard.h"    // include BobbyBoard library
#include "qfixSlaveBoard.h"     // include SlaveBoard library

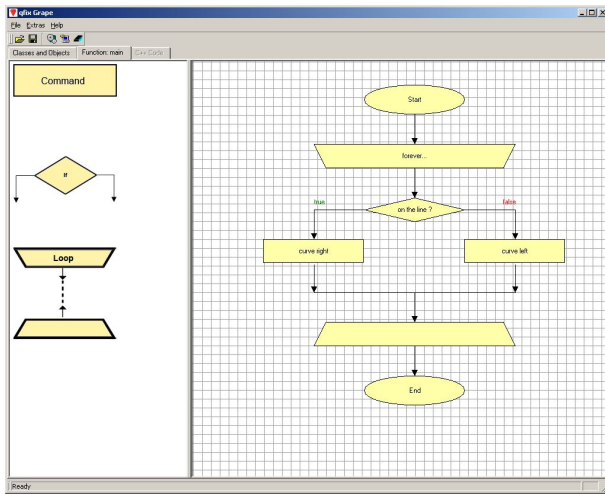
int main()
{
    BobbyBoard master;          // construct a master board object
    SlaveBoard slave0(0);       // construct a slave board object
    SlaveBoard slave1(1);       // construct a slave board object

    master.motor(0,100);        // turn on motor on master board
    slave0.motor(0,100);        // turn on motor on slave board 0
    slave0.waitForButton(0);    // wait for button on slave board 0
    slave1.ledOn(0);            // turn on LED on slave board 1
}
```

### 3 Graphical Programming Environment GRAPE

In addition to the C++ environment, we developed a new software system called *GRAPE* (which stands for GRaphical Programming Environment). With GRAPE it is possible to program the *qfix* controller boards in an object oriented way without having experience in C++.

The GRAPE application consists of three tabbed windows which are used sequentially: In the first tab, the desired classes (e.g. BobbyBoard and LCD) are loaded. Each class can then be instantiated by one or more objects. The object names can be freely chosen. The second tab holds the main window for graphical programming. Here, symbolic blocks are arranged intuitively in order to get a flow chart with the desired program flow (see Figure 9).



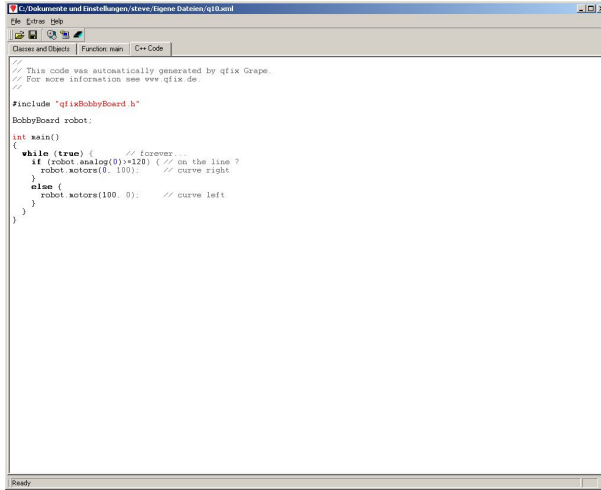
**Fig. 9.** Graphical program in GRAPE

For each symbolic icon, a properties dialog can be opened to define the semantics of the icon in a semi-graphical way: For *commands*, the user can select an object from the list of instantiated objects, then chose a method from the object’s possible methods, and then select the desired parameters from the list of possible parameters for the chosen method. This selection defines all parts of a typical object-oriented method call: `<object>.<method>(<parameters>)`.

After filling all graphical blocks with their respective meaning, the flow chart can be saved as a XML description file. This makes it possible to perform, e.g. in an individual tool, the translation to any object-oriented (or even classically procedural) programming language. In GRAPE, this translation is already integrated and the flow chart (or internally, the XML representation) is automatically translated to C++ code (see Figure 10).

With this approach, the basic concepts of a procedural programming language can easily be learned: commands, sequences of commands, if-clauses, while loops.





```

C:\Documents und Einstellungen\Steve Enderle\Dateien\qfix\src
File Edit View
Classes and Objects | Function map | C++ Code
// This code was automatically generated by qfix Grape.
// For more information see www.qfix.de

#include "qfixBobbyBoard.h"
BobbyBoard robot;

int main()
{
    while (true) { // forever
        if (robot.smlog(0)+128) { // on the line ?
            robot.motors(0, 100); // curve right
        }
        else {
            robot.motors(100, 0); // curve left
        }
    }
}
Ready

```

**Fig. 10.** Respective code in GRAPE

And, it can be studied how these concepts are translated to C++ or another programming language. In addition to that, the users learn to use given class libraries.

## 4 Experiments

In order to demonstrate the feasibility of the *qfix* parts and controller boards, we developed some typical robot and mechatronic applications.

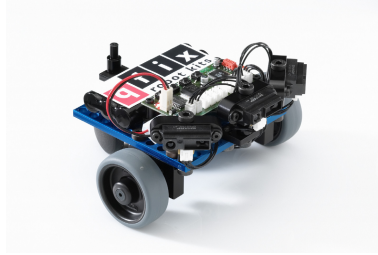
### 4.1 Differential Drive Robot

The first mobile robot is a car with two independently driven wheels and a caster wheel, all mounted on a 10cm x 10cm base plate (see Fig. 11 ). The BobbyBoard drives the two motors as well as three infrared distance sensors (Sharp GP2D120) which are used for a simple collision avoidance behaviour. An improved version also uses bumpers, a line sensor for moving along a line and an LCD for displaying messages like “front blocked” or general status information.

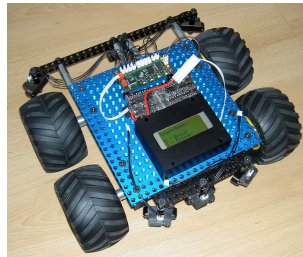
### 4.2 Offroad Robot

Figure 12 shows an “offroad” robot which was built in order to test the power of the motor controllers (L293D).

For this robot the same mainboard as in the differential drive robot above is used and drives four stronger motors, where the left and the right ones are connected in parallel. The complete platform is much bigger (main plate of 20x20cm) than the above one and includes a boxed version of the LC-display.



**Fig. 11.** Differential drive robot with two driven wheels and three IR distance sensors



**Fig. 12.** Robot arm with three DOFs



**Fig. 13.** Omnidrive platform with three omnidirectional wheels

### 4.3 Soccer Robot

As a third application, a specialized soccer robot was built in order to demonstrate the flexibility of both mechanics and electronics components. As main platform we used a round plate of about 21cm diameter with three omnidirectional wheels (see Figure 13).

In order to control the three motors, two controller boards were connected via the I<sup>2</sup>C-bus and communicate with each other to establish a reliable movement coordination. Additionally, the resulting soccer robot uses a kicker device and a so called “dribbler” to hold the ball near the robot. As sensors, infrared light sensors are used for detecting a RoboCupJunior ball. For obstacle avoidance,



**Fig. 14.** Soccer robots

infrared or ultrasonic distance sensors can be attached. The complete soccer robot including a trendy skin or “tricot” is shown in Figure 14.

## 5 Conclusion

We presented *qfix*, a construction kit for developing autonomous mobile robots and other mechatronics applications. *qfix* was mainly developed for educational and edutainment purposes. The kit consists of solid mechanical and electro-mechanical parts, powerful modular controller boards with several extension boards, and a complete C++ class library for easy support of all functionality.

Since the kits are often used in the RoboCupJunior area, where the users are only 12 or even less years old and have no programming experience, we developed the graphical programming environment GRAPE. This tool supports object oriented programming on a graphical level but directly generates C++ code which can be studied and edited.

The complete *qfix* robot kit family proves to be an appropriate tool for robot development. It is already used in educational classes and labs in schools and at universities. Additionally, the open architecture encourages the robotics community to help improving the kits.

## Acknowledgement

This work is sponsored by KTB mechatronics GmbH, Germany ([www.ktb-mechatronics.de](http://www.ktb-mechatronics.de)).

We thank Bostjan Bedenik who mainly implemented the *qfix* Grape software.

## References

1. ActivMedia: Pioneer 1 Operation Manual. RWI, Jaffrey, NH (1996)
2. Enderle, S.: The Sparrow-99 robot. Technical report, University of Ulm, Internal report (1999)

3. HandyBoard: (1998)  
<http://lcs.www.media.mit.edu/groups/el/Projects/handy-board/index.html>
4. K-Team.: Khepera – user manual (1999)
5. Kaiser, J.: Real-time communication on the CAN-bus for distributed applications with decentralized control. In: 4th IFAC International Symposium on Intelligent Components and Instruments for Control Applications, Buenos Aires, Argentina (September 2000)
6. Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E., Matsubara, H.: RoboCup — a challenge problem for AI. AI magazine, pp. 73–85 (Spring 1997)
7. Nomadic: <http://www.robots.com>
8. POLAROID: Ultrasonic Ranging System. Polaroid Corporation, 784 Memorial Drive, Cambridge, MA 02139 (1991)
9. RoboCup: <http://www.robocup.org>
10. RoboCupJunior: <http://www.robocupjunior.org/de>
11. Utz, H., Sablatnög, S., Enderle, S., Kraetzschmar, G.K., Palm, G.: Miro – Middleware for mobile robot applications. IEEE Transactions on Robotics and Automation, Special issue of on Object Oriented Distributed Control Architectures, 2002 (submitted)