Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Abstraction and Unification of Sensor Access in Mobile Robots

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Stefan Tauner
Matrikelnummer 0427514

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: A.o.Univ.-Prof. Dipl.-Ing. Dr.techn. Markus Vincze
Mitwirkung: Dipl.-Ing. Dr.techn. Michael Zillich
Dipl.-Ing. Peter Einramhof

Wien, 2014-04-23

(Unterschrift Verfasser) (Unterschrift Betreuer)

This thesis as a whole and all parts not attributed to someone else are licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Stefan Tauner
Ospelgasse 11-17/6/6, 1200 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Abstract

In the past, the development of mobile robots did not attach great importance to modular design or reusability although it is often desirable even within a single project to be able to add or replace sensors without too many consequences for other modules. Heterogeneous systems with several different busses are used to acquire sensor data which makes changes a lot harder. Also, commercial sensor modules frequently work with one bus only, support a limited number of addresses and are incompatible with hardware from other vendors.

This thesis investigates how communication with and integration of sensors in mobile robots for research and teaching can be improved.

A review of common sensor types and related work in the field of sensor access provide the foundation to choose an existing field bus to be used as a single backbone bus for a communication network in robots. A control and data transfer protocol for embedded systems is defined and implemented on top of USB. To compensate for any bus latency and jitter, a globally synchronized time base is established to correlate timestamps with all sensor values already at the source of measurement. A developer-friendly integration of the system into an example application in Robot Operating System (ROS) shows its usefulness to speed up development.

Keywords: Masters's thesis, robotics, sensors, USB, synchronization, ROS

Kurzfassung

In der Vergangenheit wurde bei der Entwicklung von mobilen Robotern oft wenig Wert auf Modularisierung und Wiederverwendbarkeit gelegt, obwohl es schon im Laufe eines einzigen Projektes oft wünschenswert wäre, Sensoren einfach hinzuzufügen oder zu tauschen, ohne, dass dies besondere Auswirkungen auf andere Module nach sich zieht. Zur Datenakquirierung werden oft heterogene Systeme mit einer Vielzahl unterschiedlicher Busse verwendet, was Änderungen und Erweiterungen erschwert: Kommerziell verfügbare Sensorplatinen sind oft nur mit einem Bus kompatibel, haben teilweise nur eine beschränkte Adressierbarkeit und können deshalb nicht einfach untereinander getauscht werden. Außerdem müssen die Zugriffe in zeitlich korrekter Abfolge durchgeführt werden und Prioritäten beachtet werden.

Diese Diplomarbeit soll Möglichkeiten aufzeigen, wie die Kommunikation mit und die Einbindung von Sensoren (und Aktuatoren) in mobilen Robotersystemen (etwa in Forschung und Lehre) vereinfacht werden können.

Ein Überblick über häufig verwendete Sensortypen bildet gemeinsam mit den in der Literatur gefundenen Lösungen die Basis, um aus den existierenden Feldbussen einen zu wählen, der als Backbone-Bus in Robotern verwendet werden kann. Ein Protokoll zur Steuerung und zum Transfer von Daten für eingebettete System wird definiert und ein Proof-of-Concept basierend auf USB implementiert. Um Einflüsse durch etwaige Latenzen und Jitter abzuschwächen, wird eine global synchronisierte Zeitbasis aufgebaut, um Messinformationen schon an der Quelle mit Zeitstempeln zu versehen. Eine Benutzfreundliche Integration des entwickelten Prototyps mit Robot Operating System (ROS) in einer Beispiel-Anwendung beweist seine Praxistauglichkeit.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem Statement . . . 2
1.3 Overview of this Thesis . . . 3

2 Transducers and their Interfaces in Robotics **5**
2.1 Sensing the Environment for Obstacle Avoidance . . . 5
2.2 Positioning & Orientation . . . 7
2.3 Range Imaging . . . 11
2.4 Actuators . . . 12
2.5 Bandwidth Requirements . . . 15

3 Possible Backbone Busses **17**
3.1 TIA-485 . . . 17
3.2 Controller Area Network (CAN) . . . 19
3.3 FlexRay . . . 23
3.4 USB . . . 26
3.5 Summary . . . 31

4 Related Work **32**
4.1 A Broader View . . . 32
4.2 Controller Area Network (CAN) . . . 34
4.3 Multi-Tier Architectures . . . 35
4.4 Morphing Bus . . . 36
4.5 Tinkerforge . . . 37

5 Methodological Approach **39**
5.1 Crosscutting Requirements . . . 39
5.2 Blueprint . . . 44

6 Synchronization **45**
6.1 Theory . . . 45
6.2 Implementation . . . 46
6.3 Results . . . 50

7 J2A — A Control and Data Transfer Protocol for Embedded Systems 51
7.1 Features 51
7.2 Implementation 54

8 Host Software Interface 60
8.1 Transducer Definitions and Firmware Interface 60
8.2 ROS Integration 63

9 Results 68
9.1 Creation of Sensor Board Firmware 68
9.2 Integration with ROS/the_missing_link 69
9.3 Test Application 71

10 Conclusion 73
10.1 Summary 73
10.2 Future Prospects 74

Bibliography I

Acronyms VIII

List of Figures X

Chapter 1

Introduction

1.1 Motivation

Acquiring and using mobile robots has become effortless thanks to a vast number of off-the-shelf robotics kits of all kinds. While these are often prepared to be extended with superstructures or other payloads like high-level sensors, changing the pre-installed (sensor) modules is usually impossible or at least very tedious. Robots built from scratch (e.g. in the course of scientific projects) on the other hand are often special-purpose and changeability is not a major goal of the design either.

It is easy to understand why this is the case after studying the architectures of robots. In Figure 1.1 a commercial robot's structure is shown. The legend of that picture sums up the main problem in a concise way: to connect all kinds of peripherals 6 different bus systems are used. This is not an uncommon configuration at all and has its roots in the (usually commercial) components used to build robots.

Mobile robots are a relatively new field in automation (e.g. compared to manufacturing processes) that often still uses experimental setups and applications. That is one reason why robotics relies on existing commercial products which were designed with other applications in mind and hence have interfaces optimized for other use cases. Robots include technologies used in cars (e.g. accelerometers which are used to trigger airbags), ships (e.g. GPS), airplanes (e.g. attitude and heading reference systems (AHRSs), smartphones (e.g. proximity sensors), video game consoles (e.g. RGB-D cameras), telecommunication networks (e.g. (wireless) Ethernet), RC models (e.g. servos) and so forth.

In research, building a modular robot from scratch is most often not an option because the focus of projects is on more advanced research questions. For example, the Automation and Control Institute (ACIN) of the Vienna University of Technology researches different aspects of robotic vision systems like object recognition, Simultaneous Localization and Mapping (SLAM) etc. [74, 20]. For experiments in real-world environments various robots are in use whose sensor configurations have sometimes to be changed. Doing so ties resources which could be better deployed focusing on the scientific work.

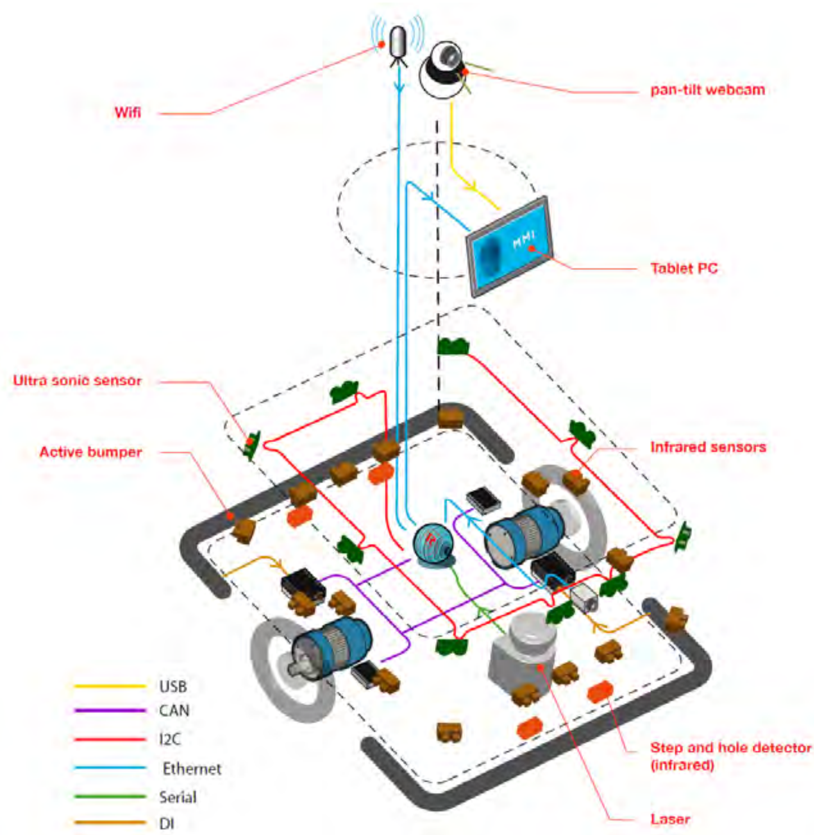


Figure 1.1: Abstract schematic of a typical off-the-self robot (ROBOSOFT robuLAB10).

1.2 Problem Statement

This section begins with a description of the mobile platform the solution is designed for. This includes the generic properties of the system but the most important parts are the sensors to connect. Based on that the problem is defined afterwards.

1.2.1 Structure, Locomotion and Power

While there are many use cases for different forms of motion the approach described hereinafter concentrates on ground-bound robots with two parallel, independently driven wheels. This form of propulsion is in widespread use due to the simplicity of construction and control, but nevertheless provides adequate mobility for (nearly) flat surfaces. The motor control loop (including reading wheel encoders) is outsourced to a custom-made daughter board whose interface can be chosen freely.

Electrical power is supplied by batteries in *sufficient* quantities with a voltage of 12 V or 24 V and are of no further concern except for that it might be desirable to measure the remaining power of the batteries.

All higher level processing is to be done on a host computer within Robot Operating System (ROS) [53] and hence all data needs to be transmitted to this host. If computation power on board of the robot is not enough to run ROS or completely solve the required algorithms then (parts of) the calculations can be outsourced to a remote computer. This is only relevant insofar as all sensor data has to be sent either to the remote host directly or via a gateway on the robot (preferably also running ROS) which is then responsible for further communication.

1.2.2 Sensors

The various sensor types will be described in detail in chapter 2, but for the sake of this chapter it is necessary to determine the maximum number and types of sensors in any possible application and to describe the overall configuration.

Table 1.1 shows how many sensors/busses of each kind should be at least accessible and how many we deem to be useful at most. The numbers are based on the existing configurations of robots in use at ACIN while paying attention to future developments and expandability. Some of these interfaces are busses which make it possible to connect more than two devices together up to different maximum limits (which sometimes have practical implications), others are point-to-point connections.

Most sensors are to be used in groups, for example, 4-8 IR sensors used at the front of a robot will usually be of the same model and only be replaced together. Sometimes coordinating sensor activation might be necessary to prevent interferences by the measuring process or to sense the environment with multiple sensors synchronously at (about) the same instant even if they are not connected to the same controller.

1.2.3 Aim of this Thesis

As explained in the motivation section sensor reconfiguration of mobile robots is challenging. Therefore, the aim of this thesis is to investigate possibilities to mitigate the problems created by the use of multiple busses in mobile robots, focusing on systems that can be used in research as well as teaching. A prototype is to be built that allows to swap (sets of) sensors quickly.

1.3 Overview of this Thesis

The second chapter gives a summary of the sensors to be handled. Afterwards some important fieldbusses that could be used to transfer accumulated sensor values are briefly discussed in chapter 3. Chapter four reviews the state of the art of communication in robots and what has been tried to unify the numerous ways to access sensors. The chapter "Methodological Approach" recaps all properties influencing the design and outlines my solution. After that, three chapters illustrate different aspects of my approach in detail. In chapter 6 a synchronization protocol described that help to overcome problems created by bus latency and jitter. The low-level communication protocol, i.e. how messages are actually formatted and transmitted is explained in chapter 7. chapter 8 details the integration with ROS while a case study of how to use the system is given in chapter 9. The last chapter sums up the work, shows unsolved problems and possible improvements.

Table 1.1: The table below shows how many busses/connections are needed for each sensor type. For each bus it shows the nominal/expected count (*nom*) as well as the maximum (*max*) of what we determined to be possibly useful. Where a maximum value is given the absence of a value in the *nom* column denotes an unknown but existing value >0, whereas 0 means that the respective sensor can be missing completely. Some types of sensors can be accessed via different interfaces (e.g. 2D Lasers do often have both USB and Ethernet sockets available). If some of these are deemed unpractical to use in our environment then they are in parentheses. For a discussion about the distinct sensor types see the respective part in chapter 2.

Bus / Sensor	**digital$_{\text{in}}$**[a]		**analog$_{\text{in}}$**[b]		**I^{2}C**[c]		**SPI**		**Serial TTL**[d]		**TIA-232**		**TIA-422**		**TIA-485**		**Ethernet**		**USB**		**CAN**	
Sensor	nom	max	nom	max	nom	max	nom	max	nom	max	nom	max	nom	max	nom	max	nom	max	nom	max	nom	max
Battery[e]			0	1	0	1																
Drives[f]							0	1	0	1												
Bumper		16																				
Cliff		16				16				(16)		(16)								(16)		
Infrared	0	8																				
Ultrasonic				16		16																
AHRS					0	1	0	1	0	1	0	1	0	1	0	1			0	1	0	1
GPS									0	1	0	1	0	1	0	1			0	1		
LIDAR[g]											0	2					0	2	0	2	0	2
ToF/RGB-D																	0	2	0	2		
Σ[h]	0	40	0	17	0	34	0	2	0	3	0	4	0	3	0	3	0	1	0	6	0	3

[a] General-purpose input/output (GPIO) pins; some of them may need to measure pulse times (see subsection 2.1.3).
[b] Analog-to-digital converter (ADC) pins.
[c] Depending on the addressing scheme supported by the sensors, it might be required to attach them to more than one Inter-Integrated Circuit (I^{2}C) master due to address conflicts.
[d] In contrast to TIA-232 interfaces found on PC peripherals, embedded systems often use a logically equivalent communication scheme but with different voltages (i.e. Transistor-Transistor Logic (TTL)) for on-board connections.
[e] To read out remaining battery power.
[f] The interface to the motor control daughter board can be chosen freely. We assume Serial Peripheral Interface (SPI) or serial hereinafter.
[g] One laser sensing forwards and backwards respectively. Due to the high bandwidth requirements the serial connections are typically used for fallback and debugging only, see section 2.3.
[h] Excluding values in parenthesis.

Chapter 2

Transducers and their Interfaces in Robotics

This chapter gives an overview of sensors and actuators commonly used in robotics with a focus on working principles and interfaces. The last section also gives an estimation of the typical bandwidth needs with a focus of the main use case discussed in this thesis.

2.1 Sensing the Environment for Obstacle Avoidance

2.1.1 Bumpers

The most basic kind of sensor in robotics are digital switches attached to bumpers. Nowadays they are almost only used to detect critical situations when other sensors have failed to detects an obstacle, but in the beginning of robotics (before that term was even coined) switches in (electro-) mechanical apparatuses were the main means of perception [58, 28, 69]. Due to their use they often need to be processed with high priority.

2.1.2 Infrared Sensors

Similar to the bumper switches infrared sensors are often used to detect dangerous situations: *cliff sensors* try to detect abrupt ground changes. They are often implemented by using binary IR distance sensors which have a much wider possible use.

The typical construction for triangulation is depicted in Figure 2.1: the emitter diode radiates (often pulsed) IR light through the outgoing lens on an object. The light is reflected by the object and enters the sensor again through a lens onto a Position Sensitive Device (PSD). The angle the light enters the sensor and hence the point on the PSD it illuminates depends on the distance between the sensor and the reflective object. Figure 2.2 shows a picture of a Sharp GP2Y0A02YK with the lenses detached.

Typical working distances are in the range of a dozen centimeters to a few meters. Sharp is basically the only manufacturer of readily available sensors. Most of their devices output the results as analog voltage in (neither linear nor proportional) relation to the detected distance.

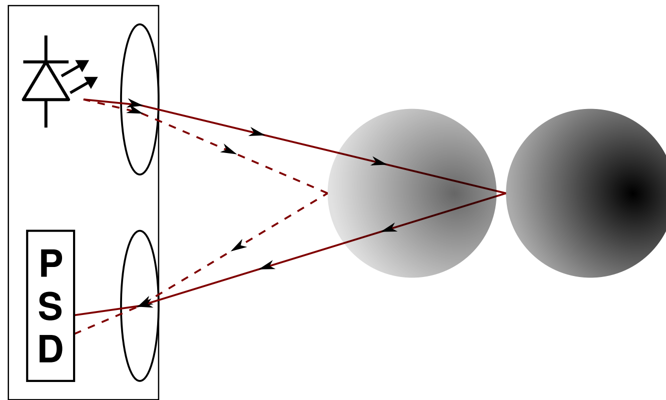


Figure 2.1: Working principle of optical distance sensors.

Alternatively there are some models that send their data serially while being clocked externally or via I^2C. The value is updated continuously at about 25 Hz.

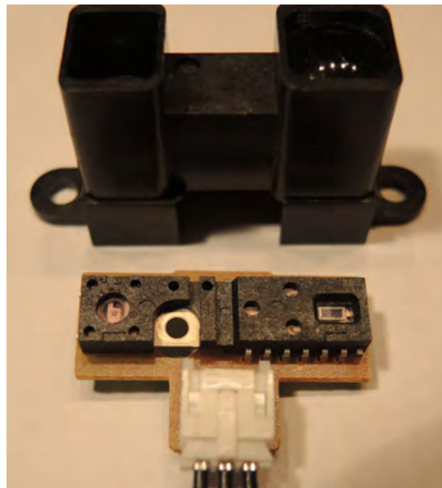


Figure 2.2: Picture of a Sharp GP2Y0A02YK IR distance sensor. The IR LED can clearly be seen on the left and the 1-dimensional PSD on the right.

2.1.3 Ultrasonic Sensors

Sonar ranging works by measuring the time of flight of ultrasonic waves sent by the sensor (cf. Figure 2.3). The commercial products available usually work around 40 kHz to 180 kHz and send multiple pulses per wave packet/measurement. Some problems of this method include the transceiver ringing continously after sending which makes it impossible to detect very near objects, the relatively wide sensitivity lobe and interference between multiple similar sensors

and due to reflections. The speed of sound depends on the temperature of the medium and has to be compensated for to achieve more accurate results [63].

The minimum range is usually about 30 cm and maximum ranges of around 10 m are possible. The distance is output as the length of a pulse, via serial (TTL, TIA-232 or Universal Serial Bus (USB) tunnel) or via I^2C. Maximum update rate for continuous operation depends on the expected maximum distance (due to echoes) and is up to approximately 25 Hz (i.e. working up to about 7 m).

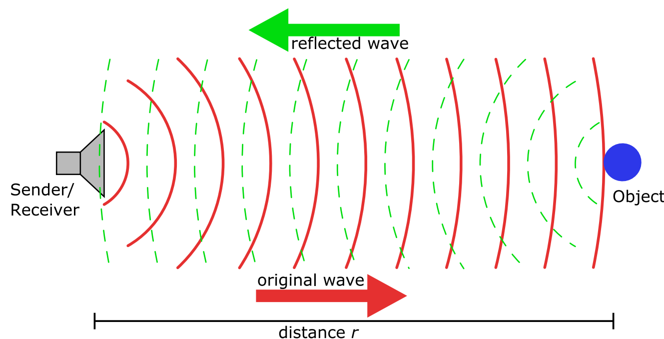


Figure 2.3: Working principle of sonar ranging.

2.2 Positioning & Orientation

Autonomous robots critically depend on knowing their own location, orientation and relative movement to be able to feed closed-loop controlling algorithms for navigation and posture adjustments. The following paragraphs give an overview of the most common sensor types used to derive the needed information.

2.2.1 Rotary Encoders

Measuring rotations allows for example to estimate the distance wheeled robots have traveled or the posture of a gripper arm. Sensors are based on wheels with encoded angular information detectable by conducting heads, optically (e.g. IR diodes and photodiodes) or magnetically (e.g. hall sensors). They can be distinguished by their outputs: **incremental** or relative rotary encoders convey only the incremental change of the rotation while **absolute** encoders output the absolute angle of the shaft — even immediately after powering up.

A very common and precise way to measure rotations of axes are quadrature encoders. They have two main output signals which are toggled alternately while the axis is rotating in one direction as depicted in Figure 2.4 making them incremental encoders.

The resolution of rotary encoders is given by their Counts per revolution (CPR) which refers to the number of full periods per 360°. As can be seen in Figure 2.4 there are four edges per period splitting one period into 4 phases. When each of these edges is taken into account (so-

called 4x decoding) then the resolution in degrees is given by

$$R = \frac{360}{4 \cdot \mathrm{cpr}} \tag{2.1}$$

Some quadrature encoders have a third signal (commonly named index) which triggers once per full rotation hence allowing to infer the absolute position of the axis from their outputs only (no external references like calibration runs needed). To ease handling quadrature signals there are separate decoder chips [5] available as well as hardware acceleration support by microcontrollers designed for motor control [4, 72].

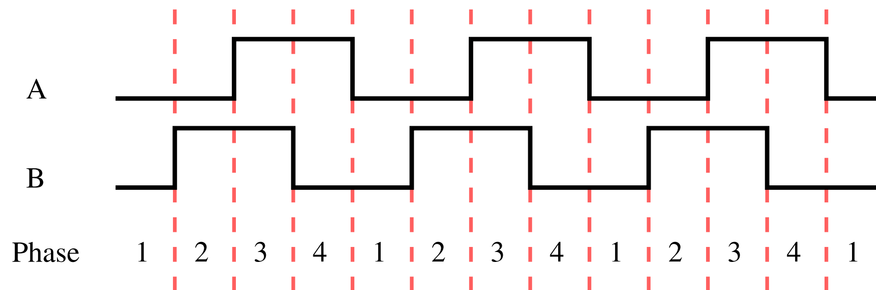


Figure 2.4: Example output of a quadrature encoder.

2.2.2 Global Positioning System (GPS)

GPS usage has become ubiquitous — even the cheapest smartphones contain a GPS receiver nowadays. Numerous documents have been written about the history, theory and implementation of GPS [9] and other similar global navigation satellite systems (GNSSs) [14] therefore I will concentrate on the few bits most relevant to the topic of this thesis.

Virtually every stand-alone GPS receiver sends its data via an ASCII-based serial protocol based on NMEA 0183 [47]. Unlike the standard which is loosely based on EIA-422 which is a differentially signaled, single-sender multi-drop bus most GPS devices use another electrical interface (e.g. EIA-232 and TTL levels over various types of wires or virtual tunnels of serial data via Bluetooth (BT), USB or other more modern busses). On top of the bit stream a standard `8n1` frame format (8 bits per frame, no parity, one stop bit) and 4800 (NMEA 0183) or 38400 baud (NMEA 0183-HS) is specified but the speed is largely ignored too. [71, 42, 65]

Blocks of useful information are combined to so-called *sentences*. These start with `$` (parametric sentences) or `!` (encapsulated sentences), followed by an address and (comma-separated) data fields, an optional checksum (prefixed with $\star$) and eventually `\r\n` (carriage return and line feed) at the end. The address field contains a talker ID of 2 bytes followed by 3 bytes specifying the data format.

The most important sentence for GNSS applications is `GGA` — Global Positioning System Fix Data:

```
$GPGGA,hhmmss.ss,llll.ll,a,yyyyy.yy,a,s,nn,x.x,x.x,M,x.x,M,x.x,xxxx*hh\r\n
```

Acquisition time of the data in Coordinated Universal Time (UTC)

Latitude in arcminutes(!), `a` is `N` or `S`

Longitude in arcminutes(!), `a` is `E` or `W`

Fix quality as numerical codes e.g. 0 no fix, 1 SPS, 2 differential SPS, 8 simulated

Number of satellites used (`00–12`)

HDOP Horizontal Dilution of Precision (lower is better)

Altitude in meters above sea level

Geoidal separation The height of mean sea level (geoid) surface above WGS-84 earth ellipsoid surface.

Age of differential GPS data in seconds

Differential reference station ID (0000-1023)

Checksum

RMC (Recommended Minimum Specific GNSS Data) and **VTG** (Course Over Ground & Ground Speed) additionally give current course and speed. NMEA 0183 allows proprietary commands (denoted by a single `P` as talker ID) which some manufacturers use to e.g. allow the baud or sentence update rate to be changed.

2.2.3 Attitude and heading reference systems (AHRSs)/Inertial measuring units (IMUs)

IMUs measure aspects of orientation in three-dimensional space. The nomenclature in scientific literature and documentation of commercial products is sometimes ambiguous: The term AHRS usually describes a device consisting of gyroscopes, accelerometers and sometimes also magnetometers (so-called magnetic, angular rate, and gravity (MARG) devices), which fuse sensor data to compensate for (accumulation of) errors and to output the combined estimation of orientation, while an IMU does output distinct (i.e. unfused) values [40]. But sometimes the terms are used interchangeably and depending on the combination of sensors, AHRSs are called 6-axis/6-DOF (w/o magnetometers) or 9-axis/9-DOF IMUs too.

Some more complex AHRSs use even GPS data or barometers additionally to MARG sensors and are often called inertial navigation systems (INSs) [68, 73]. While 9-DOF AHRSs are possible to be manufactured on a single die, many commercial products still combine multiple ICs on integrated multi-chip packages (beautifully depicted by Mikhail Svarichevsky in Figure 2.5), or even multiple packages on a board. Especially the multi-sensor devices have often a dedicated processor responsible for sensor fusion and host communication.

Gyroscopes measure the **rotation** around the three orthogonal axis in 3D space in respect to an inertial system, hence allowing to derive angular velocity and acceleration. Unlike other gyroscopes like ring laser gyroscopes (RLGs) [39] and fiber optic gyroscopes (FOGs) [67] which rely on the *Sagnac* effect [2], IMUs usually used in robotics are microelectromechanical systems (MEMSs) which are based on the Coriolis effect requiring a proof-mass to be in motion (e.g. oscillated). Important advantages of using MEMSs are small size and weight as well as reduced

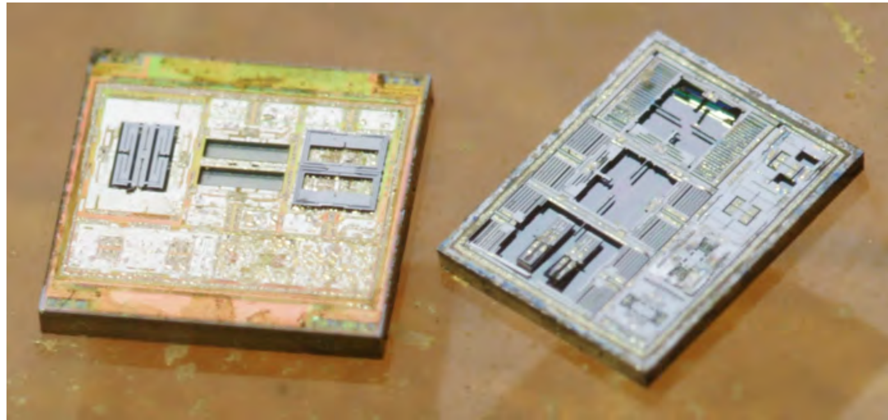


Figure 2.5: Photo of the two dies contained in an Invensense MPU6050.

costs. They perform much worse (in respect to sensitivity) than the other types mentioned above though. You can find a thorough review of modern gyroscope technology in the book *Advances in Gyroscope Technologies* [3].

Accelerometers in robotics are mostly build as MEMS like gyros and are also available in 3D configurations, meassuring linear acceleration. Various approaches based on piezoresistive, piezoelectric and capacitive effects, as well as resonators have been implemented. “All accelormeters share one common feature - the use of a seismic mass. The underlying principle of operation is that the seismic mass experiences an inertial force when exposed to acceleration, which displaces the mass from its equilibrium position.” [54] The resulting change of that displacement, which depends on the physical effects the respective sensor is based on, are then measured to derive the acceleration.

Magnetometers can be used as a compass but more importantly together with accelerometers to estimate and correct the error of gyroscope data. [40] In MEMSs they are often based on the Lorentz force, e.g. by using the Hall effect. Using the Hall effect is problematic due to the relative small sensor area with respect to the magnetic field strength to be measured. To improve sensitivity and reduce noise these issues need to be reduced by employing suitable countermeasures (e.g. using a magnetic flux concentrator). There is a broad range of other sensor techniques for different applications available as well (e.g. magnetoresistive-based sensors in data storage devices), but this is beyond the scope of this work. [59]. Recent developments often use the Lorentz force acting on a current-carrying coils by monitoring their vibration amplitude at resonance frequency [34]. To detect field distortions one can calibrate the AHRS, monitor the field strength and reduce the influence of the magnetometer values in the calculations temporarily [37] or try to compensate (at least for inclination errors) [40].

Additional data from **barometers** can help to estimate altitude [73], but suffers greatly in situations of rapidly changing air pressure (e.g. by ventilation changes) [51]. Barometers unlike other MEMSs need to be in direct contact with the measured medium (e.g. the surrounding air)

which makes their developing and especially packaging more complex. Nevertheless pressure sensors were among the first MEMSs developed in the 1970s. The common property utilized in MEMS barometers is a thin membrane sealing a cavity containing a reference pressure (e.g. vacuum). The actual pressure measurements can be based on piezoresistive, capacitive or optical (e.g. Fabry-Perot) effects monitoring the deflection of the membrane. Detecting changes of the resonant frequencies of beams which are stressed by the deformation of a membrane is also possible. [19]

The hardware **interfaces** as well as communication protocols of commercial IMUs vary a lot. There exist IMUs/AHRSs with at least these interfaces: I^2C, SPI [30], serial (TTL levels and TIA-232 [68] as well as TIA-422 and TIA-485 [71]), USB and Controller Area Network (CAN) [37].

The protocols used to query the sensor data and configure the systems are proprietary and incompatible with each other, but are usually publicly documented even if higher-level drivers are provided. Typical sensor update rates are about one to a few dozen Hertz. Often there are distinct interrupt pins available, which sometimes can be configured to indicate different events (of which the most important one signals that new IMU measurements are available). AHRSs allow to read out the estimated orientation as well as the distinct values of each axis of the various sensor types. Orientation is usually given in quaternions to avoid problems related to gimbal locks, but Euler angles or rotation matrices are often provided additionally. GPS-enabled AHRSs often provide the native NMEA 0183 sentences as well as an MARG-augmented absolute position in their proprietary protocol.

2.3 Range Imaging

Advanced visual sensors are used in robotics for both — discovering objects and self-orientation. Various techniques are employed to gather 3D data based on visual information. Explaining all their working principles and applications in detail is beyond the scope of this thesis. A basic overview of techniques is for example given on Wikipedia. The most important facts of a few selected range-finding systems are described below.

While stereo vision or even single-sensor cameras [36] allow to reconstruct the three-dimensional environment, it is rather computationally intensive. It is much more efficient to use sensors dedicated to this specific task.

If determining the distances to objects nearby *in a single plane* is sufficient (e.g. obstacle detection), then choosing light radars (LIDARs) is viable. They use light pulses of a rotating laser to sweep over the surroundings and create a depth profile of the nearest points in the scanning plane by measuring the time till the reflected light is received.

The same basic principle of using echoes is applied by Time-of-Flight (ToF) cameras. Instead of scanning an area sequentially with a single beam, they illuminate a whole scene simultaneously (usually with IR light) and measure the response with a two-dimensional sensor. This allows for much higher frame rates and hence is best suited to detect fast objects. Currently the lateral resolutions are the biggest drawback, measuring only about 200×150 pixels.

Another approach, namely using structured light, also produces two-dimensional depth maps which became known for being used by Microsoft's Kinect. A specially designed light pattern (also usually in IR) is projected into the scene and the distances are calculated from the distortions of the pattern captured on a single image.

The two-dimensional depth maps of ToF cameras or those obtained with structured light can be combined with visual images taken from a traditional camera (e.g. to easily texture 3D models reconstructed from the depth maps). This combination is then called RGB-Depth/Distance (RGB-D) camera.

In Table 2.1 the most important characteristics of a small selection of image-based range-finding sensors are compared. Attention should be paid to the last column that shows the required net bandwidth and varies greatly from below 1 Mbit/s to several hundred Mbit/s. If ordinary (high-resolution) and stereo cameras would be included the maximum bandwidth would even be higher. The bandwidth values in the table were calculated by multiplying the bit width of atomic measurement values with the value count of each scan and the frequency of scans. The value count depends mainly on the lateral resolution but there are some exceptions: Some LIDARs measure multiple echoes and/or the amplitude of the received pulses. ToF cameras usually also optionally provide the amplitude information and RGB-D images include color information. The bandwidth shown equals the complete raw information obtainable but excludes any encoding or communication protocol overhead.

Table 2.1: Summary of range-finding imaging sensors (ordered by bandwidth). Information compiled from the respective datasheets and manuals (unless noted otherwise).

Producer	Model	Type	Typ. (Max.) Range	Centroid Wavelength (λ_c)	Depth Accuracy	Lateral Resolution	Maximum Frame Rate	Max. Net Bandwidth[a]
			[m]	[nm]	[mm]		[Hz]	[Mbit/s]]
Hokuyo	URG-04LX	LIDAR	4 (5.6)	785	10 to 40	240°/0.35°	10	0.1
Sick	LMS100	LIDAR	20	905	40	270°/0.25°	50	3.5
Hokuyo	UTM-30LX-EW	LIDAR	30 (60)	905	30 to 50	270°/0.25°	40	4.7
MESA Imaging	SR4000	ToF	8 (10)	850	15	176×144 px	50	40.5
Bluetechnix	Argos3D P100	ToF	3	850	1 to 80	160×120 px	160	98.3
Microsoft	Kinect	RGB-D	5[b]	830[c]	1.5 to 40[b]	632×480 px[d]	30[e]	322.3

[a] Including potential visual/amplitude and multiecho values and excluding everything but the raw data values (i.e. ignoring any product-specific protocol headers et cetera).

[b] "The random error of depth measurements increases quadratically with increasing distance from the sensor and reaches 4 cm at the maximum range of 5 meters" [31, p. 1451].

[c] "The IR laser emitter creates a known noisy pattern of structured IR light at 830 nm" [33, p. 11].

[d] "The depth data output has a resolution of 640×480 pixels, where the rightmost 8 pixel[s] are always 'no data', so the effective resolution is 632×480 pixels" [48, p. 4].

[e] "[L]ow resolution [. . .] NIR Kinect has a frame rate of about 30 Hz" [48, p. 4].

2.4 Actuators

Most definitions of robots require them to carry out physical tasks and therefore they need to control actuators to move parts of their *body* or as a whole. The most important actuators in robotics are electrical motors which can be used e.g. to drive wheels and propellers or to move

joints of arms and legs. Below is a summary of the most important aspects of them which are documented more detailed for example in “Embedded Robotics” by Thomas Bräunl et al. [12].

2.4.1 Controlling DC Motors

To make use of motors one needs to be able to control the speed and direction of its rotation. The direction of basic DC motors can easily be controlled by using an H bridge as depicted in Figure 2.6. It usually consists of four N-channel MOSFETs, appropriate driver(s) and some auxiliary components [41]. With this configuration it is possible to drive the motor in both directions, let it run freely and brake it with only two logical inputs. To regulate the speed, pulse width modulation (PWM) can be used but the effects of alternating currents on inductive loads must be considered. Also, note that this is only an open-loop control: there is no feedback at all and any software controlling the motor has no way to determine if the robot (part) does move at all. By sensing the movement of the motor shaft for example with a rotary encoder a closed-loop control can be built.

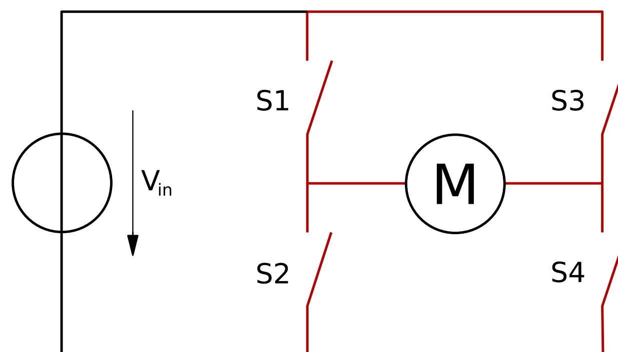


Figure 2.6: Schematic of an H bridge.

2.4.2 Stepper Motors

Stepper motors are popular for more sophisticated applications, like precise control of gripper arms [26]. Their rotors have two notched wheels of soft magnetic material attached to the two poles of a permanent magnet as can be seen in Figure 2.7 (on the right). The coils of the stator are aligned in pairs so that the teeth of their pole shoes align perfectly with the notches of the rotor and the magnetic field ($\vec{B}$) can flow through them with minimal resistance. While this pair is energized (A in Figure 2.7 the notches of the other, inactive stator pair are aligned exactly *between* the rotor teeth. To rotate the shaft, this other pair has to become magnetized while the previous one is deactivated. Depending on the polarity either the dark notches corresponding to the south pole or the light gray teeth belonging to the north pole are then attracted to the pole shoe teeth. Such a small rotation is called a step. Due to the four steps needed to complete a whole cycle of magnetizing the two pairs with both polarities a complete revolution needs $n \cdot 4$ steps where n is the number of teeth of one of the notched wheels.

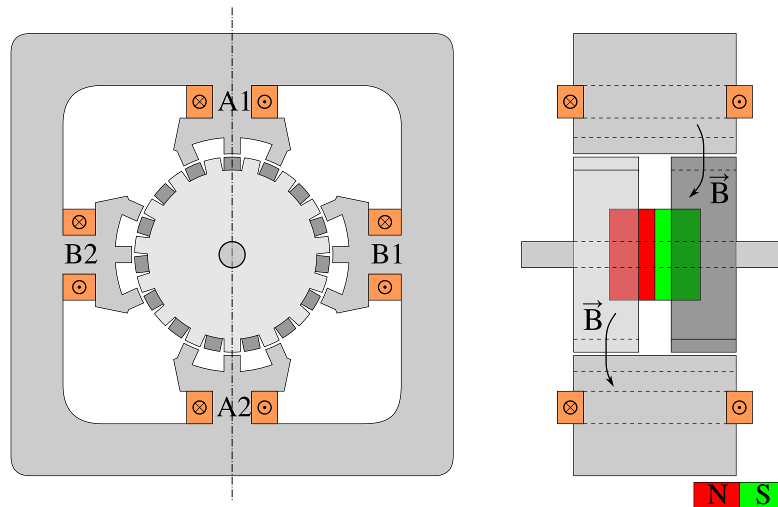


Figure 2.7: Schematic cross section of a stepper motor.

Although there can be fairly many teeth in a stepper motor running it one full step at a time makes a quite unsmooth rotation. To overcome this the coils of different pairs can be partially magnetized concurrently by intermediate currents. For example if both $A1$ and $B1$ are equally polarized so their notches become north poles the wheel will move into a position where the light gray teeth are equally away from the teeth of $A1$ and $B1$. By modulating the current through the coils even more positions are possible. Driving a motor this way is called microstepping and leads to a much smoother rotation as well as resolution. To ease handling of stepper motors especially when driven by microstepping there are complete driver chips available that rotate a motor about one step on rising edges of a control signal and include the necessary H bridge as well [1].

Having such a precise control over the rotation may seem like making stepper motors a perfect fit for any application that would normally require additional sensors to form a closed-loop control, but this ignores the fact that the process described above is idealized. If the torque of the motor is not high enough to move the rotor for example because the motor drives a wheel of a robot which is blocked by a wall then (groups of) steps might be missed without the control algorithm able to notice. If the odometry process takes the rotation of the wheel for granted then it will determine a wrong location (similarly to when the data of rotary enoders are used although some wheels are spinning without traction).

2.4.3 Servo Motors

If you combine an ordinary DC motor with a gear box and a feedback control and limit its rotation to about $180°$ then you get a servo motor. Their only input (beside power) is a PWM signal with a period of $20\,\text{ms}$ ($50\,\text{Hz}$) whose duty cycle specifies the angle of the rotor. The neutral position is always assumed when signal is high for $1.5\,\text{ms}$ per cycle but the extreme values (for the duty cycle as well as the respective angles) depends on the manufacturer and

model but are usually about $\pm 90°$ and ± 1 ms from the center. An idealized version is depicted in Figure 2.8.

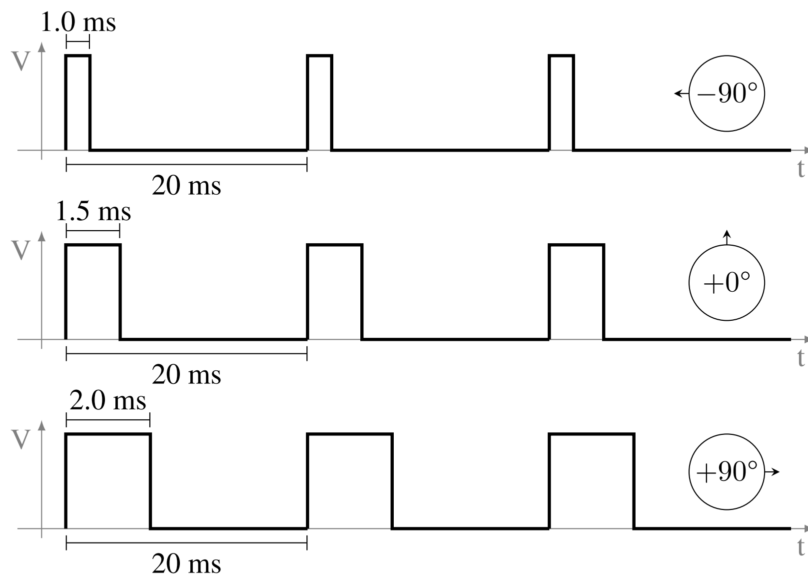


Figure 2.8: PWM control of a servo motor.

2.5 Bandwidth Requirements

So far this chapter described mainly the working principles of sensors and actuators typically used in mobile robots. This section will focus on their bandwidth requirements.

As shown in the previous sections the representation of sensor data as well as arranging their creation and transmission can be quite variously. Some sensors deliver their information whenever new data is available (e.g. rotary encoders) which needs to be processed instantly to not lose any values which is most vital for incremental data. Others need to be polled for updates or even starting new measurements.

Encoding of the data itself as well as communication protocol overheads are another important aspect of bandwidth estimation. Table 2.2 abstracts those away as much as possible to show raw information amounts only. The table proofs the wide spread of bandwidth requirements in systems of mobile robots.

While bandwidth is important another characteristic of communication is just as significant, namely latency. Without context it does not make sense to rate its influence though and hence it is discussed in detail later.

Table 2.2: Estimation of maximum useful bandwidths of single sensors based on bit widths of atomic measurements and sensible acquiring periods.

Type	Atomic Data Size	Frequency	Bandwidth
	[bit]	[Hz]	[bit/s]
Bumpers	1	irregular	<1
Cliff	1	irregular	<1
Infrared	≥ 8	15 to 50	$\geq$(120 to 400)
Ultrasonic	12 to 16	25	300 to 400
GPS (approximate entropy of GGA sentence)	132	10	1320
AHRS/IMU (raw data of gyro-, accelero- and magnetometer)	16×3 $+ 16 \times 3$ $+ 13 \times 3$	50	6750
LIDAR (raw values of 1 to 3 echoes pre pulse)	(16 to 18) $\times$ (1 to 3) $\times$ (600 to 1000)	10 to 50	100×10^3 to 5×10^6
ToF	(20 to 25) $\times 10^3$ $\times$ (1 to 2) $\times$ 16	50 to 160	40×10^6 to 100×10^6
RGB-D	$640 \times 480 \times 24$ $+ 632 \times 480 \times 11$	30	$\geq 300 \times 10^6$

Chapter 3

Possible Backbone Busses

This chapter shortly summarizes various bus systems potentially capable of being used as a backbone bus in a mobile robot. Some busses which might seem to be principally suitable for the task were not reviewed in detail because they obviously do not fit the requirements:

Ethernet While there are some attempts to make use of Ethernet as a (real-time) fieldbus (e.g. TTEthernet (SAE AS6802), AFDX (ARINC 664), PROFINET, EtherCAT, Sercos III), they require customized (i.e. expensive) hardware but even standard Ethernet controllers are not very common in microcontrollers (yet).

Time-Triggered Protocol (TTP) None of the TTP variants suits the requirements: TTP/A is too slow, inflexible and complex; TTP/C is additionally more or less end-of-life (EOL); TTEthernet is too expensive (see above).

Local Interconnect Network (LIN) A complementary bus to CAN for use cases where the latter would be too expensive to implement. It supports speeds up to 20 kbit/s only and hence is too slow.

3.1 TIA-485

TIA-485 (more commonly known as RS-485) describes the physical layer (only) of a differentially signaled, half-duplex bus normally using one twisted-pair cable. “Cable routing is generally a daisy chain [. . .] although some systems may allow stub cabling from a main backbone cable.” It allows for speeds up to a few dozen Mbit/s for distances of a few meters down to a few hundred kbit/s at up to 1 km. Drivers need to support 32 unit loads that is 32×12 kΩ which represents “receivers, drivers in the passive state, and any common-mode loading presented by the termination” (at the full -7 V to 12 V common-mode voltage range). Therefore the actual number of supported receivers on the bus depends on the attributes of the devices involved [64].

Various upper-layer protocols can use TIA-485 or slight modifications thereof as a base, many of which are proprietary fieldbusses for industrial, building automation etc. (e.g. Profibus, DMX512, IEC 61107).

3.1.1 MODBUS

Initially invented as communication protocol for programmable logic controllers in 1979, MODBUS is still widely used as industrial field bus. The MODBUS specification is separated into two layers, where the single upper layer is called "MODBUS application layer" (hereinafter *application layer*. There are currently three bottom layers specified [46]:

MODBUS "over serial line" This specifies the transport of the application layer over TIA-485 (half-duplex with 2 wires or, optionally, full-duplex with 4 wires) and TIA-232 [45]. The description below will focus on the combination of this variant with the application layer.

MODBUS TCP/IP Allows to send MODBUS Protocol Data Units (PDUs) via TCP/IP. The specification allows for gateways translating between MODBUS TCP/IP nodes and other kinds by embedding the node's address in an additional header [44].

MODBUS PLUS A proprietary "high speed token passing network" [46].

When using a serial connection there is "only one node (the master node) that issues explicit commands to one of the "slave" nodes and processes responses. Slave nodes will not typically transmit data without a request from the master node, and do not communicate with other slaves." The master can also broadcast information to all nodes, but then no responses are allowed.

Two transmission modes are defined: Remote Terminal Unit (RTU) and ASCII. The default is to use the RTU mode which uses common 8 bit serial frames with (by default) even parity (called characters within the specification). The whole messages need to be separated by a silent bus time of at least 3.5 characters as shown in Figure 3.1.

Start	Address	Function	Data	CRC	End
≥3.5 chars	8 bits	8 bits	0-252 bytes	16 bits	≥3.5 chars

Figure 3.1: RTU message format.

The second (and optional) mode is called ASCII mode which uses two 7 bit frames to encode 8 bit values as ASCII-encoded hexadecimals. In that mode each message has to start with a colon (`:`) and end with a sequence of carriage return and line feed (`\r\n`) as depicted in Figure 3.2.

Start	Address	Function	Data	LRC	End
1 char :	2 chars	2 chars	up to 252×2 chars	2 chars	2 chars CR, LF

Figure 3.2: ASCII message format.

Both types use different checksum algorithms to detect errors. RTU uses a 16-bit cyclic redundancy check (CRC) whereas ASCII mode uses an 8-bit longitudinal redundancy check (LRC). The address field identifies the target slave, where a value *0* indicates a broadcast and the

range of *248–255* are reserved, allowing for up to *247* slaves. The physical layer specification follows TIA-485 and guarantees only up to 32 nodes but, depending on the bus load, allows for more. Alternatively, one can extend the maximum number of nodes by using an active repeater [45].

In the application layer specification the nodes initiating a transaction are called *clients* and the responding nodes are denoted as *servers*. The *function* field of the request defines the action the server has to fulfill. If need be, further bytes (of the data field) can provide more information on that action, e.g. sub-functions, starting addresses, or lengths. The server responds with the same function code as received in the request on success or sets the first bit which is reserved for this to indicate an *exception*. In the case of failure an *exception code* can be sent to the client indicating the cause of the error (e.g. address out of bounds).

The main data model of MODBUS is based on the four table types summarized in Table 3.1. The tables can be thought as a kind of mapping between the MODBUS interface provided by the server and the internal representation of I/Os etc. "The distinctions between inputs and outputs, and between bit-addressable and word-addressable data items, do not imply any application behavior. It is perfectly acceptable, and very common, to regard all four tables as overlaying one another, if this is the most natural interpretation on the target machine in question."

Table 3.1: MODBUS data model (adapted from the MODBUS specification [46]).

Table Name	Object Type	Type of Access
Discretes Input	Single bit	Read-Only
Coils	Single bit	Read-Write
Input Registers	16-bit word	Read-Only
Holding Registers	16-bit word	Read-Write

In Table 3.2 the publicly specified function codes are represented but there exist code ranges reserved for user-defined functions (i.e. 65–72 and 100–110). There are various function codes for reading and writing single or multiple entries in the primary tables. These use starting addresses of 16 bit allowing to handle tables up to 65536 rows. The functions to access *files* allow to combine multiple reads or writes of potentially non-contiguous addresses into a single request. The usefulness of this is rather limited due to the payload limit of modbus messages (i.e. 252 B).

There are also some diagnostic functions defined which can return error counters or disable (and re-enable) all MODBUS communication. The MODBUS Encapsulated Interface (MEI) is concept that allows other protocols to be (limitedly) tunneled as MODBUS' payload. The only two protocols currently specified are a simple (string-based) device identification method and CANopen (see section 3.2.1 for more details about the latter) [46].

3.2 Controller Area Network (CAN)

CAN is an asynchronous multi-master serial shared-medium data bus with widespread use in the automotive industry (among others). "A CAN bus is intended to be as close as possible to a single line structure", but multiple unterminated stubs of a few meters can be handled [62].

Table 3.2: MODBUS public function codes (adapted from the MODBUS specification [46]).

				Function Codes	
				Code	**Sub Code**
Data Access	**Bit Access**	Physical Discrete Inputs	Read Discrete Inputs	02	
		Internal Bits or Physical coils	Read Coils	01	
			Write Single Coil	05	
			Write Multiple Coils	15	
	16 Bit Access	Physical Input Registers	Read Input Register	04	
		Internal Registers or Physical Output Registers	Read Holding Reg.	03	
			Write Single Reg.	06	
			Write Multiple Reg.	16	
			R/W Multiple Reg.	23	
			Mask Write Reg.	22	
			Read FIFO queue	24	
	File record access		Read File record	20	
			Write File record	21	
Diagnostics			Read Exception status	07	
			Diagnostics	08	00–18,20
			Get Com event counter	11	
			Get Com Event Log	12	
			Report Server ID	17	
MODBUS Encapsulated Interface (MEI)			CANopen	43	13
			Read device ID	43	14

One of the most distinctive features of CAN is its special kind of collision handling, sometimes called Carrier Sense Multiple Access with Collision Resolution (CSMA/CR) (there is no consistent naming in the literature). It relies on the property of the physical layer to distinguish dominant and recessive bit values. Recessive bits are not actively driven by the transmitter but use the dormant state of the bus (e.g. the voltage level applied by pull-up resistors). The dominant values are enforced by the sender and practically overwrite recessive values potentially sent by other devices at the same time. This concept is used to encode priority of messages and stop the transmission of messages with lower priority when a collision is detected before it leads to invalid data: Each sender observes the bus while transmitting the arbitration field at the start of each messages and compares the bus's state with sent values. If it detects a discrepancy because one of his recessive bits was overwritten by another sender it backs off and retries later. Note that the transmission of the higher priority message is completely unaffected by the collision. [55]

While this has obvious advantages in a network scheme with fixed priorities, the need to immediately detect collided bits limits the data rate and/or communication distances: The signal wave has to have enough time to propagate from one end of the bus to the other and back within a single bit's detection time frame. The maximum length of networks with a bit rate of 1 Mbit/s is about 40 m (depending on the medium and interface hardware used) [50].

The value used for arbitration is also used to uniquely identify (and hence address) messages;

there is no other way to systematically distinguish messages nor are there sender and receive addresses. Version 2.0B of CAN defines an extended identification field of 29 bit additionally to the standard format of 11 bit in prior versions. All messages are protected by a CRC and can contain 0–8 data bytes. Bit stuffing is used after five consecutive bits of the same polarity.

The designated successor of CAN, named CAN with Flexible Data-Rate (CAN FD), enables higher throughput by allowing to switch to an alternate (higher) *Data Bit Rate* after the arbitration phase in the middle of the control field (i.e. after the Bit Rate Switch (BRS) flag) and by supporting up to 64 payload bytes. The data rate is switched back in the CRC phase (namely after the first bit of the CRC delimiter field). The frame format was changed for CAN FD and two new CRC algorithms were added that are used depending on the number of data bytes [56]. Combined with the extended ID variant of version 2.0B (which is also available in CAN FD) this makes frame lengths quite variable as depicted in Figure 3.3.

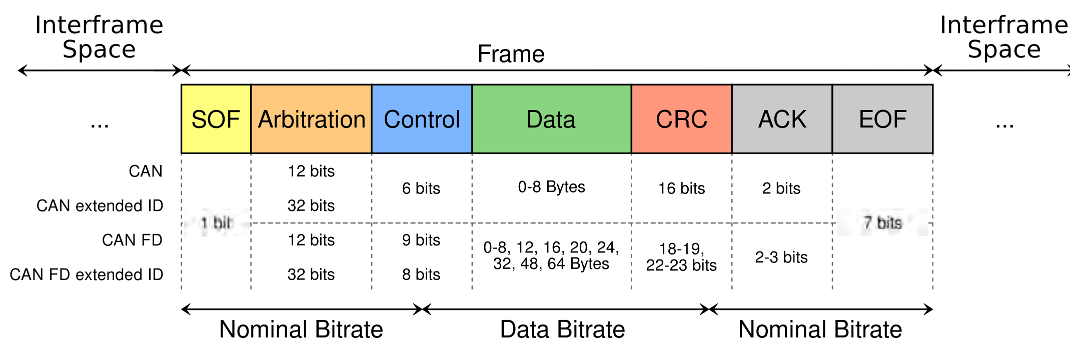


Figure 3.3: Frame formats in CAN.

CAN does not specify upper layer protocols, so network designers need to implement their own (or use one of the numerous extensions, see next section). This also implies that the scheduling of messages, especially in real-time systems, need to be done manually/on top of CAN without any support apart from the priorities expressed by the message IDs. This is a hard problem as shown by the fact that it took the scientific community over 10 years to figure out a flaw in an analysis algorithm used throughout the industry for schedulability analysis [13].

3.2.1 CAN Extensions

Due the limited scope of the basic CAN protocol there is a wide variety of protocols on top of it. Below two of them are summarized, namely TTCAN and CANopen.

Many of the protocols are focused on niche applications which led to overlapping goals and unnecessarily repeated efforts, for example, before TTCAN was standardized, a number of protocols (e.g. CAN Kingdom, CANaerospace) incorporated features to allow hard real-time applications. Other examples of higher level protocols based on CAN are MilCAN (for military vehicles), NMEA 2000 (marine industry), ISO 11783/ISOBUS (agriculture), or SAE J1939 (automotive industry mainly in USA) but there are many more open[1] as well as proprietary

[1] Here “open” means that anyone is allowed to buy the documentation and license for a specific protocol—it does by

standards.

Time-Triggered CAN (TTCAN)

TTCAN adds a cyclic transmission schedule similar to the working principle of TTP (cf. TTP's rounds and cluster cycles [49, section 5.2.1] with basic cycles and matrix cycles in TTCAN). All TTCAN nodes synchronize on reference messages, sent at the beginning of each cycle by one of several predefined nodes, i.e. the time masters. Redundant time masters use CAN arbitration as explained above to establish a hierarchy and determine the current master, i.e. the one sending with the highest priority. The uniform cycles allow all other nodes to recalibrate their local clocks and compensate for drifts between them and the time master. It is also possible for the time master to synchronize the scheduling to external events by delaying the start of one cycle till the event happens. All nodes keep track of their local offsets to the time values received in the reference messages and can therefore establish a global time base usable by applications [35].

CANopen

Another protocol based upon CAN is CANopen which spreads over many documents published by CAN in Automation (CiA) and sometimes picked up by other standards bodies (e.g. CiA 301 became EN 50325-4). CiA 301 *Application layer and communication profile* specifies requirements of interfaces between controllers, working conditions etc. Every node contains an Object Dictionary (OD) which is essentially a table with subtables, which describes the device and also stores current process data (measurements). Each entry in the OD does have a data type attached which are also defined within the OD. There are several "basic" (pre-defined or user-defined) types as well as "complex" (cf. C structs), composed types. In order to provide configuration and simulation software the needed information, OD specifications as well as the configuration of respective nodes can be stored in EDS.

There are different ways to transfer information in a CANopen network. Communicating with **Service Data Objects (SDOs)** involves a request for a (sub) table entry and the respective response. For data larger than the few bytes available in a single CAN frame, segmented transfers are possible by first sending the number of bytes to transfer and setting a dedicated bit. This can be used, as its name indicates, for configuration and diagnostics.

Process Data Objects (PDOs) can be used for more common (simplex) process data transfers . Activity of the nodes in this mode depends on the transmit trigger configuration, which can mandate event- or time-driven activation as well as polling by other nodes. The latter can even be synchronized among multiple devices when the polling node sends a `SYNC` Special Function Object (SFO) and also works for writes (e.g. allowing multiple actuators to apply some value synchronously). Other SFOs contain time stamps or emergency error codes. The last type of messages are Network Management Objects (NMOs) which are used for network management (e.g. coordinate boot-up or reset individual nodes).

CANopen devides the 11-bit CAN IDs into 4 bit function code and 7 bit node IDs allowing for 127 different nodes to be addressed. The node IDs can be set physically on the device or by

no means imply public availability of any documentation.

using Layer Setting Services, CiA 305 (LSS). The function code corresponds to one "communication entry" in the "Communication Profile Area" of the OD (index `1000h-1FFFh`) which define the mapping between the function codes and respective communication parameters. "The predefined connection set defines 4 Receive PDOs, 4 Transmit PDOs, 1 SDO (occupying 2 CAN IDs), 1 Emergency Object and 1 Node-Error-Control Identifier. It also supports the broadcasting of non-confirmed Network Management (NMT)-Module-Control services, SYNC- and Time Stamp-objects." [10]

The 29 bit identifiers of CAN Version 2.0B are only rudimentary used if available [32].

The CiA standards define different aspects which can be grouped into 3 major categories:

Frameworks
: are the lowest layer of CANopen. They are based directly upon CAN and have document numbers starting with 3 (e.g. DS-301).

Device profiles
: ensure interoperability by specifying the details of a device type (e.g. how the OD is set up).

Application profiles
: define the communication patterns of a whole application domain (e.g. DSP-417 covers lift control systems).

Henk Boterenbrood wrote an excellent more detailed summary of CANopen while working on the ATLAS detector control system later to be used at CERN [10]. An even more thorough discussion can be found in the book "Embedded networking with CAN and CANopen" [52].

3.3 FlexRay

When it became clear that future applications in cars demand more bandwidth and features than could possibly delivered even by multiple CAN busses, a consortium was formed by major automotive and electronics corporations to conceive a new protocol: FlexRay. Version 3 was later also laid down in ISO standard 17458 and is in widespread use in automobiles today.

3.3.1 Protocol Overview

FlexRay supports time-triggered as well as a event-triggered communication by using a time-division multiple access (TDMA) scheme that reserves time slots for static and dynamic data. Communication is divided hierarchically into recurrent cycles with a static and an optional dynamic segment followed by a symbol window (also not mandatory) and the network idle time (NIT) at the top level. The two segments are split into static slots and minislots as shown in Figure 3.4. The main difference between them is that the former, as the name implies, uses a static scheduling where nodes need to send a frame in each of their designated slots even if they don't have relevant data to transfer and no additional time may be used. In the dynamic segment, on the other hand, a priority-based scheme is used: The earlier dynamic frames have a higher

priority and the associated node is free to use multiple (subsequent) minislots to extend the dynamic frame as needed. The Dynamic Trailing Sequence (DTS) makes sure that every dynamic frame ends exactly at the border of two minislots with enough time for all nodes to recognize the end of transmission.

The symbol window does not have any arbitration specified and is used for network maintenance functions like coordinating wakeups of nodes during operation. Normally nodes synchronize at startup when the leading node sends a Collision Avoidance Symbol (CAS) after listening for two full cycles without any communication. It continues then with its predefined schedule of cycles but sends one *startup* frame per cycle instead of normal frames with payload. "Due to the frame ID, which corresponds to the slot ID and the value of the cycle counter which is also part of the frame, the listening coldstart nodes know the time of the leading coldstart node." [49, p. 136f.] After observing two cycles, these nodes can perform rate correction of their clock and are allowed to participate after another two cycles in which they double-check their synchronization. The NIT at the end of each cycles allows the nodes to stay in sync later on by giving them time to recalculate deviations and adjust their timing.

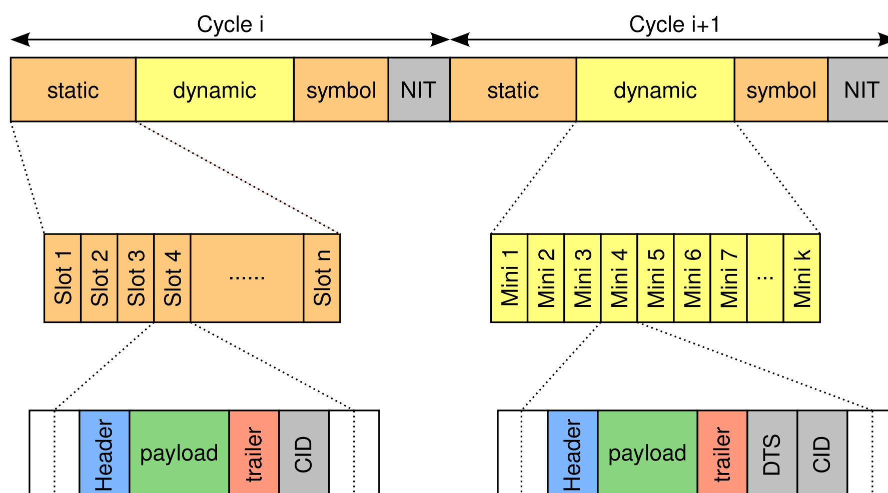


Figure 3.4: Structure of FlexRay communication.

3.3.2 Frame Format

The FlexRay frame format is depicted in Figure 3.5. The bits at the beginning are used to designate the various special frames as described above. The definition of the header CRC field is remarkable for two reasons. The FlexRay specification demands the header CRC to be precomputed to further enhance protection against faulty CC. This works because the covered bits of the header are known beforehand for the static segment: the node IDs as well as the payload length and the other bits are configured at configuration time.

But what about the dynamic segment? It uses the same frame format, but the length of the payload may vary [49, p. 128]! This makes it impossible to use constants for the CRC values of dynamic frames. The specification does not elaborate on that at all [22, sections 4.2.8 and

4.5.2], but the patent filed at the European Patent Office on FlexRay does give a hint: "Among its other uses, the Header CRC field of a FlexRay frame is intended to provide protection against improper modification of the Sync Bit field by a faulty communication controller (CC). The CC that is responsible for transmitting a particular frame shall not compute the Header CRC field for that frame. [...] This is possible because, *for fixed length frames at least,* the Header CRC would be constant." [8] [emphasis added]. It is unclear to me why it is sufficient for dynamic segments to let the host processor, or the communication controller (CC) itself, calculate the CRC values while for the static segments hard-coded CRC values are a must. In any case, the specification is not precise in this regard which is never a good sign.

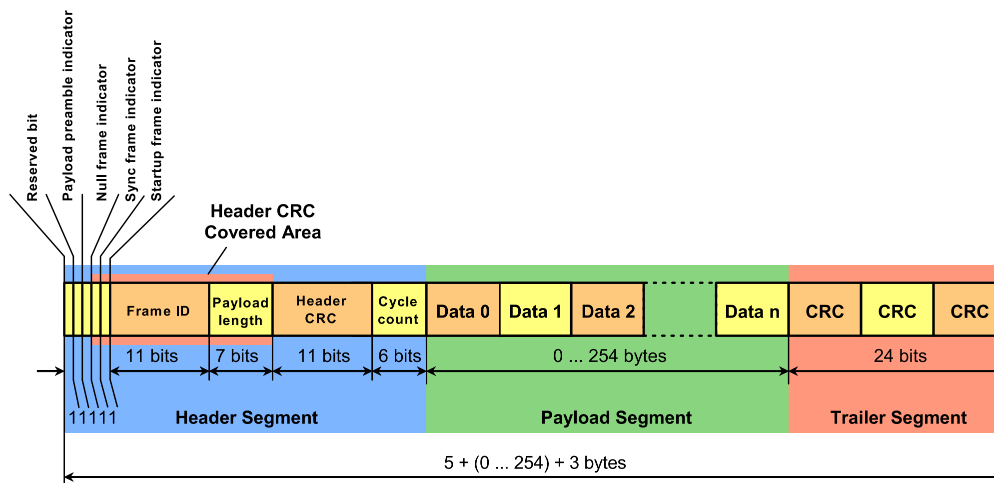
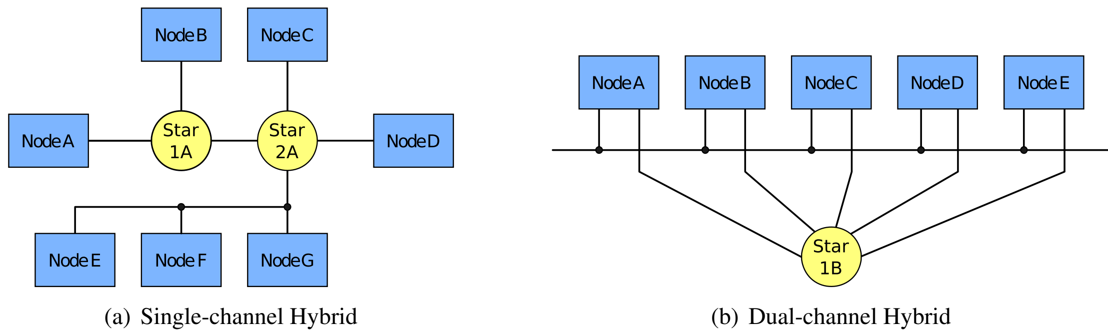


Figure 3.5: FlexRay frame format.

3.3.3 Physical Layer and Cluster Topology

FlexRay 3.0 supports bit rates of up to 10 Mbit/s on each of two independent physical channels, called *A* and *B*, which allow for redundancy and/or increased bandwidth (but nodes can also be attached to one channel only). Each channel may contain busses without closed loops and stubs - even full passive stars, one active star hub (or up to two cascaded star hubs for bit rates of 5 Mbit/s or less) making complex topologies possible like those in Figure 3.6 [21].

Bus guardians are mechanisms that determine if a sender is faulty and then restricting its access to a communication bus [27]. FlexRay offers two forms of bus guardians; both are optional. They can be implemented inside the bus driver module of each FlexRay node or within an active star hub. Bus guardians can monitor the correct issuing of messages and if they contain an indepedent clock they can do so even in the time domain.



(a) Single-channel Hybrid (b) Dual-channel Hybrid

Figure 3.6: FlexRay topology examples.

3.4 Universal Serial Bus (USB)

USB is undeniably the most prominent bus reviewed in this work shipping in billions of devices (e.g. smartphones, PCs) each year. It was invented to replace so-called legacy busses to connect peripherals (input and output devices like keyboards or printers) with PCs in the mid-1990s. It has evolved since then among other things by increasing gross bitrate over 800-fold (from 12 Mbit/s in USB 1.0 to 10 Gbit/s in USB 3.1) which allowed its use in more bandwidth-demanding applications like external data storage. The available bandwidths in USB have dedicated denotations summarized in Table 3.3.

Table 3.3: USB: Evolution of gross bitrate.

Name	Bitrate	Introduced in version…	Issue Date
Low-speed	1.5 Mbit/s	1.0	1996-01-15
Full-speed	12 Mbit/s		
High-speed	480 Mbit/s	2.0	2000-04-27
SuperSpeed	5 Gbit/s	3.0	2008-11-12
SuperSpeedPlus	10 Gbit/s	3.1	2013-07-26

Some features and properties are specified by supplements and addendums to the main specifications (often in the form of engineering change notice (ECN)). Hereafter focus will mainly be on version 2.0 of the specification since this is currently the predominant type in low-cost and embedded devices and the features following it are not essential for the topic of this thesis. The information compiled below was mainly extracted from the specification [66]. Where this was insufficient *USB Complete* [6] filled the gaps and clarified uncertainties.

3.4.1 Physical Layer

Unlike most other busses discussed here (with PoE-enabled Ethernet probably being the only exception), it does also provide limited power to the attached devices. Like bandwidth, the

available power was raised over the years driven by the increasing need to charge wireless devices quicker or attach more powerful devices without requiring additional power supplies and cables.

A single master denoted *root hub* controls transmissions on the bus. As the name suggests, the topology used in USB is a rooted tree where the (slave) devices are located at the leaves. The inner nodes are called hubs and are active units involved in controlling communication. The maximum allowed height of the tree (i.e. the maximum number of links from the root hub to a device) is six, that allows for up to five hubs between the root hub and a device. The hubs are able to translate between links with different speeds which allows to share a fast upstream link between mixed-speed downstream links without slowing down the faster device(s).

Cables and connectors are specified in the standard as well. Cable lengths are limited by properties like signal attenuation and end-to-end signal delay resulting in a maximum length of about 5 m (and 3 m for low-speed links) between any two nodes. They contain four wires for ground, 5 V (V_{BUS}) and one pair of wires to transfer data differentially (D+and D-). Any but low-speed links require the data wires to be twisted and an overall shield. There are two categories of connectors divided by whether they are facing up- or downstream (i.e. in direction of the root hub (type A) or end devices (B)).

The data is encoded non-return to zero inverted (NRZI) (i.e. 0 is represented by a change of the signal level and 1 by no change at clock boundaries) and bit stuffing is used (after six '1' bits) to ensure adequate transitions for synchronization. Two data wires allow to encode 2 digital states additionally to those used for regular data bits named Single-ended 0 (SE0) (D+and D-low) and Single-ended 1 (SE1) (D+and D-high). SE0 is actually used and involved in device state handling and signaling End-of-Packet (EOP), while (steady) SE1 signals are not allowed. The specification partially abstracts the physical state of the bus away by using variables j and k to denote the other two states in which D+and D-are driven to diverge. This allows to describe the signaling uniformly although actual voltage levels differ in Low- and High-speed.

3.4.2 Framing

In Figure 3.7 an overview of the framing used in USB is depicted. Bus time is divided into fixed time *frames* of 1 ms for Low- and Full-speed links and $\frac{1}{8}$ of that (i.e. 125 µs) for High-speed. Every frame begins with a Start-of-Frame (SOF) packet which is used for synchronization and also contains a 11 bit frame ID (0–7FFh). The frame ID is incremented every 1 ms (even on High-speed resulting in the transmission of the same ID 8 times). At the end of each frame there is a period of idle time when no device (not even the root hub) is allowed to send. This End-of-Frame (EOF) interval guarantees that the next SOF packet can be sent without interference.

Like every packet the SOF contains a SYNC field at the beginning, then a Packet ID (PID) which defines the type of the packet followed by the actual payload (i.e. the frame number, cf. Figure 3.7). Most packet types ensure integrity of their payload by adding a CRC (either 5 bit or 16 bit) before appending the mandatory EOP.

Between SOF and EOF the host is free to schedule transactions which are divided into up to 3 phases: Token, Data and Handshake. The packets sent in each phase have a common layout respectively as shown in Figure 3.8. The Token packet starts a transaction by addressing exactly one device and informing it about the art of the transaction. In data transactions there follows

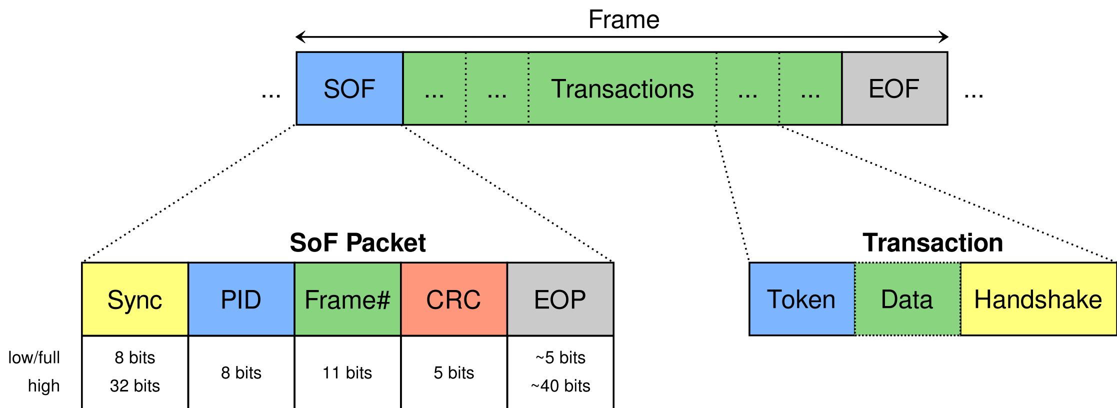


Figure 3.7: USB framing.

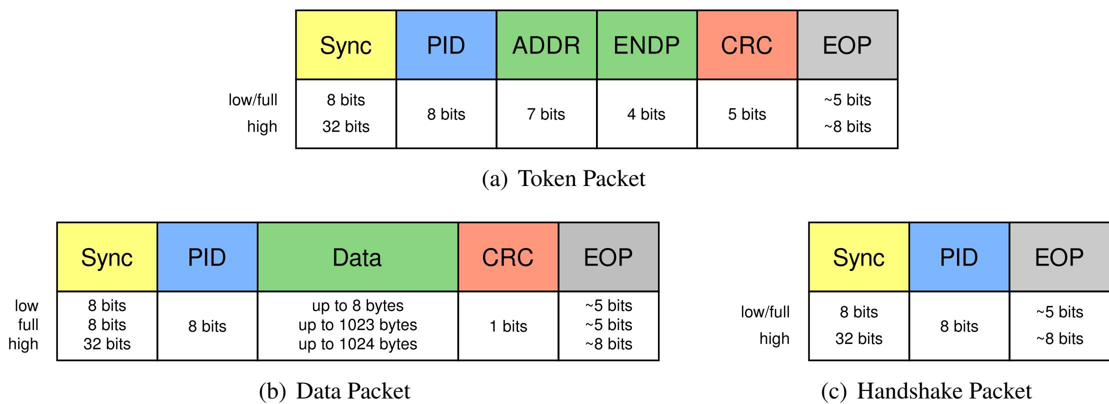


Figure 3.8: Main types of USB packets.

a Data packet, either sent by the host immediately after the Token packet or as a reply by the device. To ensure data sequence across multiple transaction there are multiple Data packets defined which are basically sent alternately. At the end the receiver replies with a Handshake packet to report the status of the transaction which can be used for error recovery and flow control. All possible PIDs are listed in Table 3.4 with a short description.

3.4.3 Endpoints and Pipes

All communication on USB is send along virtual *pipes* which connect the host (software) with an *endpoint* on the device. Endpoints are addressed by its endpoint number (in the range of 0–15) and the direction of the pipe which is denoted *IN* if payload data is sent to the host and *OUT* if the data originates at the host. Pipes are unidirectional and are set up for distinct transfer types that accommodates different demands on communication:

Table 3.4: USB PID types (adapted from the USB 2.0 specification [66]).

Type	PID[a]	Name	Description
Reserved	`0000`		
Token	`0001`	OUT	Address for transfers from host to device.
	`1001`	IN	Address for transfers from device to host.
	`0101`	SOF	Start of frame.
	`1101`	SETUP	Address for control transfers from host to device.
Data	`0011`	DATA0	Even-numbered data packet.
	`1011`	DATA1	Odd-numbered data packet.
	`0111`	DATA2	Data packet for high-bandwidth isochronous transfers.
	`1111`	MDATA	Data packet for high-bandwidth isochronous and split transfers.
Hand-shake	`0010`	ACK	Receiver accepts error-free data packet.
	`1010`	NAK	Receiving device cannot accept data or transmitting device cannot send data.
	`1110`	STALL	Endpoint is halted or a control pipe request is not supported.
	`0110`	NYET	No response yet from receiver.
Special	`1100`	PRE ERR	(Token) Low-speed preamble. Enables downstream bus traffic to low-speed devices. (Handshake) split transaction error.
	`1000`	SPLIT	(Token) High-speed split transaction.
	`0100`	PING	(Token) High-speed flow control probe for a bulk/control endpoint.

[a] Shown with the most significant bit (MSB) first. Transmitted on the wires with the least significant bit (LSB) first, followed by its one's complement.

Control transfers are used to setup and control devices. Control endpoints support transferring data in both directions and hence consist of an IN and OUT endpoint with the same endpoint number.

Bulk transfers make sure that all data is received correctly but may be slowed down due to schedulding of more important messages.

Interrupt transfers have a maximum latency between communication attempts. The device tells the host how often it should check for new data.

Isochronous transfers are used for constant-bitrate data that does tolerate occasional errors like media streams.

3.4.4 Device Architecture and Enumeration

After physically attaching a device, the enumeration process is started when the hub which connects the device reports this upstream. The host then assigns the new device a unique 7 bit address (allowing for a maximum of 127 devices on a single USB). Endpoints of a device are addressed by consecutive numbers starting with zero where support for the default control endpoint at address 0 is mandatory. This is used by the host to read out tables called descriptors to determine various properties and capabilities of a device and to configure it.

An overview of how the descriptors (and therefore devices) are organized is depicted in Figure 3.9. It starts with the *device descriptor* which describes properties pertaining to the whole device like vendor and device IDs. Before any endpoint other than the default control endpoint can be accessed the device has to be configured, i.e. the host needs to select one of the *configurations* mentioned in the descriptors. Configuration 0 is special as it is only active in the unconfigured, so-called *Address state* where the device got its unique address already but only endpoint 0 is reachable. High-speed devices have two additional descriptor types to outline differences if the device is operated at the alternative speed: *device_qualifier* for device-wide information and the *other_speed_configuration* descriptor(s) which have the same structure as normal configuration descriptors. Consequently these devices may theoretically have about twice as much configurations as a single-speed device.

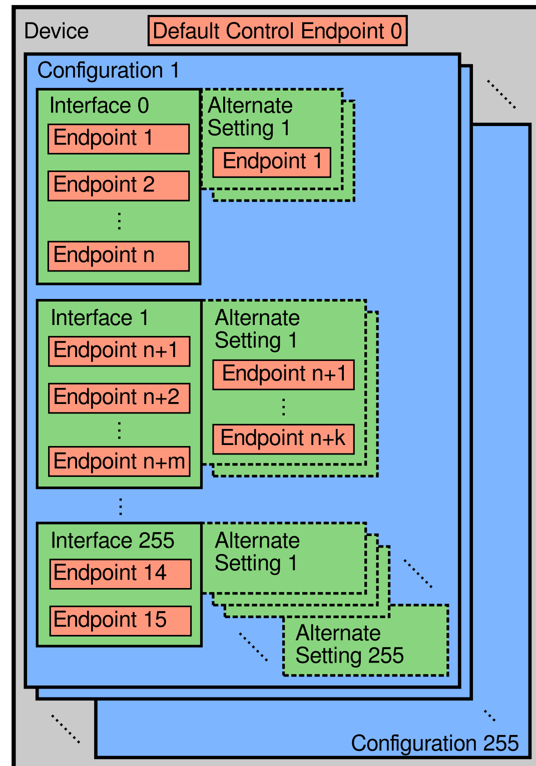


Figure 3.9: USB device architecture.

To ensure that all pipes required for a single feature or application are available concurrently, endpoints can be bundled in *interfaces*. Each interface can have one or more *alternate settings* which allows the host to select alternative options, e.g. if the default (i.e. alternate setting 0) requires too much bandwidth to be reserved in the schedule. The alternate settings theoretically do not have to have the same number or types of endpoints and therefore influence the addresses of the subsequent endpoints because they are not allowed to share addresses even with currently inactive endpoints.

Most maximum counts of entities mentioned above and as depicted in Figure 3.9 originate from the 8 bit-wide data fields/indices applicable in messages sent to the default control endpoint. The exception is the number of endpoints which is limited by the 4 bit available for them in Token packets.

Device Classes

The device and interface descriptors can specify a device/interface class by setting the so-called class, sub-class and protocol fields accordingly. These can be used to denote devices/interfaces which comply to some extended specification, e.g. mass storage devices. This allows unified software drivers to be used which do not rely on device ID whitelists. A value of 0 in the device descriptor class field indicates that each interface describes its class independently which makes unified drivers possible even for *composite devices* (i.e. devices with multiple independent interfaces).

3.4.5 Bandwidth Rationing and Scheduling

One important fact of USB is, that scheduling of messages is not predefined but is evaluated by the host each time a pipe is set up to check for schedulability. The descriptors characterize the maximum bandwidth of endpoints by denoting its maximum packet size as well the maximum interval the endpoint should be polled for new data. To guarantee the various properties of the different transfer types a catalog of rules is applied when deciding schedulability of a new pipe. The most important rules restrict the percentage of reserveable transfer time for interrupt and isochronous endpoints as well as guarantee a certain part of it for control transfers. Remaining slots can then be used by any transfer type including bulk pipes.

If the requirements specified by all endpoint descriptors of a configuration are fulfillable then requests to select this configuration are granted. The actual scheduling of messages on the bus happens dynamically depending on the actual requests from the host software, the rules for bus rationing etc.

3.5 Summary

The busses reviewed above all, have various advantages and disadvantages when considered as a potential backbone bus for mobile robots. This will be discussed in detail in chapter chapter 5 but before that the next chapter will take a look at the status quo and state of the art of busses in mobile robots reported in literature.

Chapter 4

Related Work

Investigating the state of the art of modular bus systems in embedded systems and especially mobile robots within the scientific community showed that the issue is largely ignored and outside the focus of most projects.

4.1 A Broader View

There are some niches of interest that are working on related topics which produced some interesting papers in that domain.

4.1.1 Modular robots

One such outstanding community yielding ideas worth mentioning, is working on modular (reconfigurable) robots. They have to tackle multiple problems, for example:

- Coordinating docking of elements w/o a stable, pre-configured communication bus
- Dynamically forming of new communication nets after docking

Beside the mechanical and algorithmic challenges of connecting modules securely and making them cooperative, the need for system-wide communication has to be satisfied. An obvious approach would be to use radio, but there are also robots organizing the attachment of themselves by transmitting data via IR LEDs which can be continued to be easily used after docking without electrical issues [60]. Connecting together electrical inter-module busses takes more effort (due to the handling of switching, termination and topologies) [24]. Thanks to the short distances within such modules and the relative simple and few sensors per robot these approaches can mostly avoid dealing with sensor networks directly.

4.1.2 Machine-Readable Datasheets

On top of the communication medium the actual sensor data to be transferred needs to be formatted in a predetermined way to be used by the receiver. To do this without a pre-defined

configuration of the complete system it is necessary to propagate not only the attachment of sensors and the availability of data, but also information regarding capabilities, value ranges, units and tolerances etc.

The terms electronic data sheet (EDS) and *smart transducer* are inseparably associated with IEEE 1451 and describe the combination of sensing/actuating elements, signal pre-processing (conditioning, amplification, digitalization, conversion) and a communication interface including machine-readable datasheets. The IEEE 1451 family is separated into about ten interrelated standard documents of mixed approval state which have been developed by different working groups. Figure 4.1 tries to coherently depict the responsibilities within an abstract IEEE 1451-conformant system, while Table 4.1 sums up the state and contents of the standards.

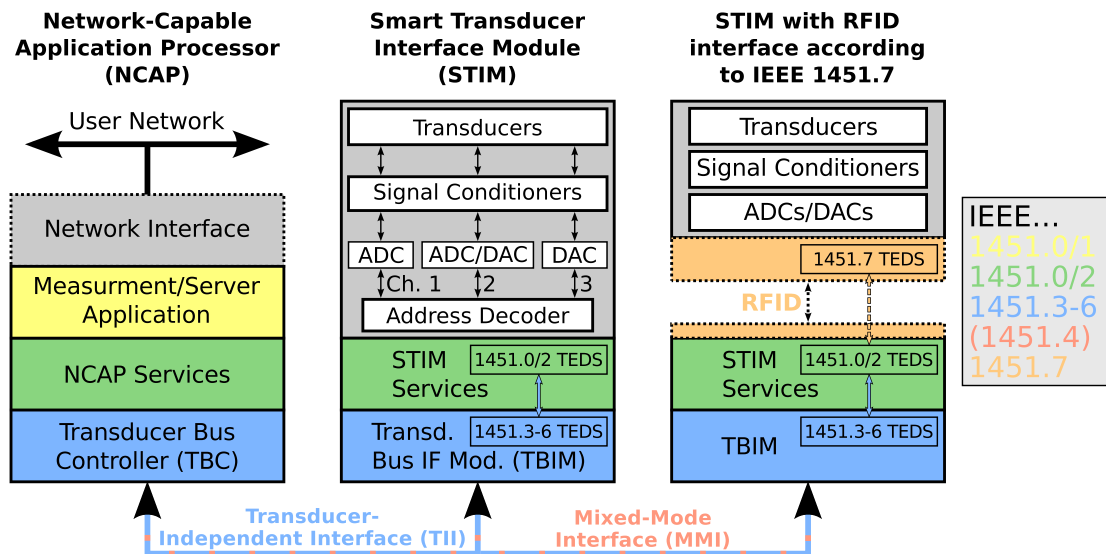


Figure 4.1: A *simple* overview of IEEE 1451.

Similar to the well-known OSI model the structure of IEEE 1451 is divided into layers. In contrast to the former the borders between layers are rather indistinct and many definitions and properties are shared. A IEEE 1451 system contains 2 major elements:

- The **Network-Capable Application Processor (NCAP)** either executes a local measurement application or provides a gateway function for remote applications (e.g. supervisory control and data acquisition (SCADA) systems) accessing the NCAP through an arbitrary user network.
- The **Smart Transducer Interface Module (STIM)** comprises the actual transducers including signal conditioning etc. and the interface module which allows an NCAP to communicate with it.

One of the most distinct features of IEEE 1451 are Transducer Electronic Data Sheets (TEDSs) which provide a machine-readable description of various properties of a STIM in a standardized

format. This includes general information on the STIM (e.g. a unique identifier), regarding individual transducer channels (e.g. physical units, uncertainty or calibration data), and also the physical connection between the STIM and the NCAP. For this connection there are various options defined in the standard documents, like point-to-point, multi-drop, wireless or even a mixed mode transferring analog and digital data on the same wires. Theoretically the NCAP and STIM can form a single device, but on the other hand even the STIM can be partitioned into an interface and transducer block. The latter case is specified in IEEE 1451.7 where RFID transmissions are used to access the transducer(s).

IEEE 1451 is clearly a very complex bundle of specifications building upon numerous other standards and technologies. Part 6 of the suite describing the use of CAN and CANopen as a NCAP-to-STIM interface was never finished and part 4 specifying the use of HomePNA 2.0 to form a multidrop bus between an NCAP and multiple STIMs has been withdrawn. It is therefore not very surprising that IEEE 1451 has not gained much popularity outside academia (yet) although the *Internet of things* has been hyped for over a decade. One notable implementation of an STIM uses USB for the user network [15], which was later even integrated with a USB-to-Ethernet gateway to form a complete IEEE 1451-compatible instrumentation bus [16].

Table 4.1: Summary of IEEE 1451 standards.

Name	State[a]	Description
1451.0-2007	Active	Common functions, communication protocols, and TEDS formats for interoperability within the IEEE 1451 family.
1451.1-1999	Active	*NCAP* information and processing model
1451.2-1997	Active	*Transducer to NCAP* communication (initial standard to define this interface)
P1451.2/D20	Proposed	Serial point-to-point communication (renewal leveraging features of IEEE 1451.0 and limiting scope to the actual connection based on SPI)
1451.3-2004	Withdrawn	*Distributed multidrop systems* (based on HomePNA 2.0, i.e. ITU-G.9951 and G.9952)
1451.4-2004	Active	*Mixed-mode communication* by adding a digital bus (based on the 1-Wire protocol) to an existing analog interface
1451.5-2007	Active	*Wireless communication* (based on Bluetooth, ZigBee and IEEE 802.11)
1451.6	Unknown[b]	*CANopen communication*
1451.7-2010	Active	STIM to Transducers communication via RFID

[a] As of 2014-02-26 according to `http://ieeexplore.ieee.org/xpl/standards.jsp?queryText=1451`

[b] There definitely was a draft standard for IEEE 1451.6 once but since then there was apparently not much public activity. Working group (WG) 6 was even removed from the list of WGs at `http://www.nist.gov/el/isd/ieee/ieee1451.cfm` and the draft is also not listed on the IEEE website.

4.2 Controller Area Network (CAN)

Without any doubt CAN is capable of transmitting the messages to successfully control robots. It was first proposed to use CAN in this field as early as 1996 by Warui and Rachid [70]. Interestingly, this paper also cites the paper with the flawed schedulability analysis mentioned in section 3.2. Since then numerous robots described in the scientific literature have adopted this approach and use one or even multiple CAN busses.

4.2.1 Multiple Busses on a Biped Robot

One example for this is a biped robot using four CAN busses connected to a PC/104-*Plus* board via two dual-channel cards (each costing about €250) [7]. The x86 CPU on that board runs a realtime-enhanced version of Linux for "Walking Trajectory Generation and Control" and to connect an operator via WLAN. A total of 13 motor controllers communicate with the main board via four CAN busses. The commercial motor drivers used are fully integrated with a CAN interface and allow to attach limited peripherals (e.g. switches) and execute user programs. This local processing capability helps reducing bus communication and local fault reaction times. Most other sensor data is sent by two dedicated CAN-enabled microcontrollers with the exception of an IMU which is directly connected to the PC/104-*Plus* board via USB.

4.2.2 Splashelec: an Autopilot for Sailing Boats

A much simpler (and cheaper) configuration than those described above, is used by a French team to build an autopilot for a sailing boat [11]. Their goal is to automate many physical tasks needed to operate a sailing boat so that even disabled persons can easily control it without manual trimming or steering. All sensors, actuators and processing elements share one CAN bus. Because most of these peripherals lack a native CAN interface the project is using Arduino-compatible CAN shields (costing less than €40) in combination with a customized microcontroller board as a gateway. They deliberately decided against a complex high-level protocol and defined their own protocol on top of plain CAN. The boat is equipped with hardware keys and a joystick, or an Android-based tablet (connected via BT to a CAN node) for user interaction.

4.2.3 CANopen

As explained earlier, CANopen can ease the creation of communication system based on CAN.

One project exploiting this successfully was very briefly described in a paper in 2004 [25]. The Robotics Bus exploits various CANopen features, e.g. using common 8P8C modular connectors for transferring data and power, heart beats, as well as PDOs and the OD. It was tested in two applications without a central master but no evaluation or comparison to possible other implementations was given.

A more sophisticated approach was taken by a German team that also includes power distribution but tries to provide complete building blocks for mobile robots that are connected by a backplane [43]. They did not completely migrate to CAN for all busses though, e.g. the embedded PC is connected via USB. No exact details about the interfacing are given but that mostly AT90CAN128 were used. An example configuration can be seen in Figure 4.2.

4.3 Multi-Tier Architectures

In robots without a single backbone bus, often a layered approach is used.

One such example that was elaborately documented in a master thesis is Universal Robot Bus (URB) [61]. It uses I^2C busses to attach up to 16 data acquisition nodes to central bridges, that are able to connect to the main control board over TIA-232 and USB (or similar interfaces).

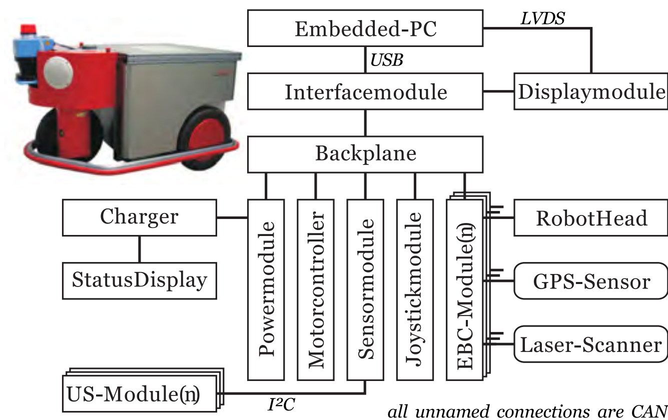


Figure 4.2: Schematic of one example using the adaptable architecture presented in [43].

Normally the CPU requests data from a node by sending a request to the associated bridge which then relays it to the addressed node and forwards the reply back to the main board. But it is also possible to periodically read out sensor values without intervention from the host CPU which only receives the data updates. The respective bridge coordinates the polling for new values instead after being set up by the host once. Another important feature is that all nodes of one bus can be synchronized to a heartbeat sent by the respective bridge.

URB lacks any error detection or recovery: neither CRCs nor sequence numbers are implemented. Combined with I^2C which also lacks any such mechanism (apart from a possible bus failure recognition depending on the controller implementation) this can easily lead to communication errors. Presumably this led to the decision to migrate to CAN in the long term[1] which was never realized according to public information.

4.4 Morphing Bus

A (yet) rather exotic way to create easily reconfigurable bus systems is to use an FPGA.

Byung Hwa Kim has worked on this topic in his PhD thesis by developing the Morphing Bus [18]. It uses so-called wedge PCBs that accommodate the electronics for sensors and actuators. Each of them contains one upstream and one downstream connector that connect the wedges with each other and the base board as depicted in Figure 4.3. The first few pins are used for power and are just passed on by all wedges. The data pins are different as each wedge routes the first pins to its own modules and shifts the remaining pins from the upstream port to the begin of the downstream pin right after the power pins. This allows the following wedge again to take the first data pins and use them for its own purpose.

The scheme of routing explained above makes the pin mapping of the connector on the base board and hence that of the FPGA depending on the order of the wedges. Also, there are no bus

[1] At least the department offered such a project to students citing increased reliability as the goal: `http://bdrl.ceng.metu.edu.tr/_media/projects/ug_summer/bdrl-up-18_urb-can_bus.doc`

arbiters or guardians involved. Both facts make it necessary to preconfigure the system statically before powering up and make plug-and-play impossible. The advantage is that the signals are directly routed to the FPGA pins and one can generate interface modules appropriate for the peripheral in question without a shared bus which completely eliminates the need for extra bus logic to do for example framing, segmenting or message scheduling.

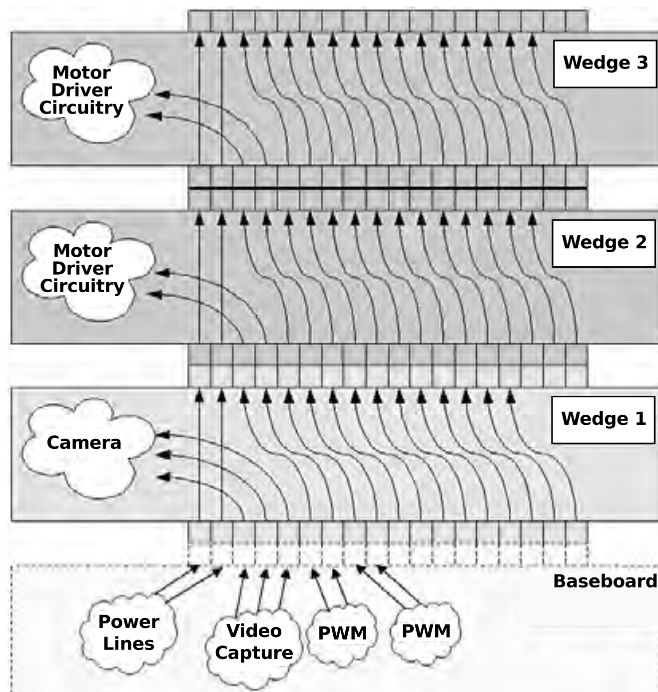


Figure 4.3: Diagram of the connection principle of the Morphing Bus.

4.5 Tinkerforge

Unlike all other projects described in this chapter Tinkerforge (`http://tinkerforge.com`) is not of academic but commercial origin. Its purpose and implementation is quite related to the topic of this thesis and justifies its analysis herein, which is based on its public documentation, accompanying forum and some personal correspondence with one of the founders (Bastian Nordmeyer).

The name Tinkerforge refers to a system of stackable boards as well as to the company based in Germany that produces them. The system consists mainly of two components: stackable building blocks called *Bricks* and smaller boards called *Bricklets* extending the functionality of Bricks, e.g. by providing sensor data.

Tinkerforge (still[2]) requires the user application to run on a separate host, normally con-

[2] Making stand-alone applications possible is planned by offering a Rapid Embedded Development (RED) Brick running Linux on an ARM Cortex-A7 or A8 [38].

nected via USB to a so-called *Master Brick* which controls the communication between all bricks of a whole stack. With *Master Extensions* it is possible to use other connections too, e.g. WLAN, TIA-485 (utilizing Modbus RTU, cf. subsection 3.1.1), Zigbee or Ethernet (even with Power over Ethernet (PoE)). It is even possible to bridge between two stacks (which is called cross-linking) using TIA-485 or Zigbee without any changes in the user application.

Bricks are PCBs of $(4 \times 4)\,\mathrm{cm}^2$ in size featuring an ARM Cortex-M3 from Atmel's SAM3S family. They can be stacked by two 30-pin connectors on each side of a Brick's PCBs. One plug pair is used for power distribution and sensing, while the other provides data connections. Power sensing is only possible when a dedicated power supply Brick is used which is only needed if the power supplied via USB is insufficient. The data connector accommodates two SPI busses including numerous chip select lines, one $\mathrm{I^2C}$ and a serial bus, and also several support signals. One of the SPI busses is used to communicate with a maximum of 8 other Bricks while the serial and $\mathrm{I^2C}$ bus, as well as the second SPI bus are dedicated to connect the master brick to up to two Master Extensions. Routing of the chip select lines works similar to that of the Morphing Bus as depicted in Figure 4.3 which essentially auto-configures the connections on attachment.

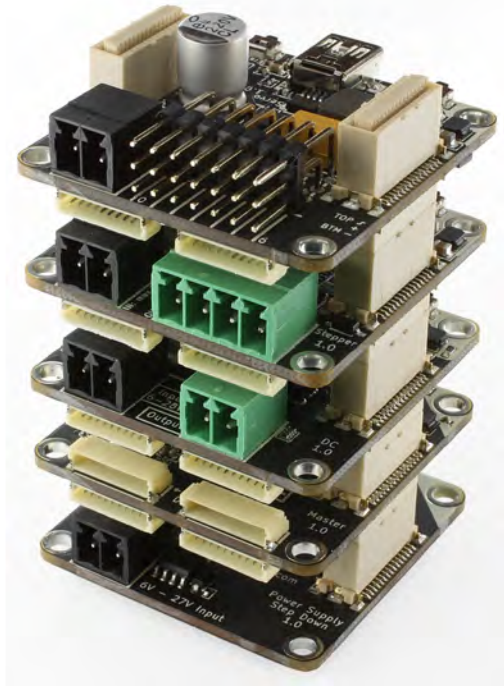


Figure 4.4: A stack of Tinkerforge Bricks.

Additionally Bricks can contain up to four Bricklet connectors with 10 pins each. They supply power, one $\mathrm{I^2C}$ bus and some I/O lines and a chip select to each Bricklet. Unlike Bricks, Bricklets have no fixed size. They are used to measure and control various physical values like voltage, temperature, light, rotation etc., or interface Bricks with more complex devices like a GPS receiver or different LCDs. Each Bricklet includes an EEPROM that can store position-independent code which gets preloaded by the respective Brick at startup for later execution. These so-called plugins can be utilized by the user program via the Tinkerforge API.

The API lets user programms easily access the features of Bricks and Bricklets. There exist bindings for all major programming language and even a python script that allows to communicate with a Tinkerforge system interactively on a Unix shell or by a script. The bindings relay any API function call to a TCP/IP connection. The remote end of that connection is normally a local daemon (i.e. `brickd`) which forwards it to the appropriate Master Brick via USB. For stacks containing an Ethernet or WIFI Extension it is also possible to directly connect to the Master brick.

All software provided by Tinkerforge is licensed under GPL version 2 or later, and the hardware under the CERN Open Hardware License. Design descriptions of all PCBs can be downloaded as KiCad files.

Chapter 5

Methodological Approach

While the previous chapters are meant to build a base of understanding of the underlying topics, this and the following chapters will describe the contributions of this thesis. The first part of this chapter recapitulates the requirements considering the information of the previous chapters and presents early process of the project including the rationales behind the decisions made. The second part outlines the approach taken and includes an overview of the hardware and software components (to be) used in the prototype.

5.1 Crosscutting Requirements

The general considerations regarding the mobile robot were discussed in section 1.2 with a focus on the number of different sensors needed (cf. Table 1.1). The various transducer types are portrayed in chapter 2 which concludes with additional information regarding the respective data amounts and frequencies in section 2.5. In chapter 3 various communication busses are reviewed and chapter 4 shows how others dealt with the problem at hand. Based on that, this section describes the preliminaries of the decision to use a single backbone bus as well as its selection.

5.1.1 One Board to Rule Them All?

The first approach was to look for a single embedded board, ideally hosting an x86 CPU (to easily run non-experimental ROS and software only available for x86), which could provide all necessary interfaces. Unfortunately there are not many x86-compatible embedded designs on the market and most have significant draw backs. The bifferboard (`http://www.bifferos.co.uk`) for example has low performance and way too few peripherals for the prospected task.

The only SOCs possibly up to the mark are the Vortex86 family from DMP Electronics Inc. They contain an almost complete x86 computer including a 32 bit core clocked at up to 1 GHz, a north- and southbridge and even embedded flash. It provides many peripheral interfaces that are very uncommon on normal x86 computers like PWM or CAN. Additionally, DMP does also produce SBCs named “RoBoard” based on the Vortex86DX SOCs. The RB-110, for example, features 256 MB RAM, a 6 V to 24 V DC/DC converter, a MicroSD and a Mini PCI slot, an

8 channel 10 bit ADC (AD-7918), 6 serial ports (1×TIA-232, 1×TIA-485, 4×TTL level), 16 PWM channels, 3×USB, 1×I^2C and one Fast Ethernet port. All of this in a convenient packages of about $(10 \times 6 \times 2)\,\mathrm{cm}^3$ [17].

The major issue prohibiting the use of any Vortex86-based machine to connect all possible transducers of a mobile robot is that many functions of the SOC are multiplexed, e.g. the SPI pins are shared with some GPIO and PWM pins. Setting the multiplexers is done in software but usually they are statically configured by the firmware under control of the manufacturer not the customer.[1] In some applications these aspects might not be a problem, but it certainly reduces the versatility of the chip.

The search for a non-x86-based board that would at least provide all the necessary interfaces was not successful either and hence the original approach of using just one board was reconsidered.

5.1.2 A Single Bus to Rule Them All

Instead of using a single board, it was decided to choose a single (type of) backbone bus to connect an x86 host PC to daughter boards. These boards in turn handle the interaction with clusters of sensors independently and act as gateways between the backbone bus and the various sensor busses. To select a proper bus system for this, a number of attributes need to be evaluated.

Bandwidth

An important aspect is the net bandwidth actually available for communication. We can roughly estimate an upper bound for it by combining the number of sensors as listed in Table 1.1 with their bandwidth needs as summarized in Table 2.2. As Table 5.1 shows, there is a tremendous gap between the imaging sensors and everything else: All non-imaging sensors combined require a little more than 20 kbit/s, while the least bandwidth-hungry imaging sensors, namely a pair of Hokuyo URG-04LX, need five times that much and the possible range goes up even to over half a gigabit per second for two Kinects (or similar RGB-D cameras). The total range of bandwidths covers therefore about 5 orders of magnitude(!), but one needs to keep in mind that these are maximum values. For example, if amplitude values of ToF cameras are not needed by the application then the amount of data sent by them is halved. A versatile platform has nevertheless to support both configurations of course.

Traditional fieldbusses provide much less and even state-of-the-art automotive busses like FlexRay support no more than about 10 Mbit/s of gross bandwidth. To connect the non-imaging sensors alone on the other hand, even CAN would suffice which is also confirmed by the numerous examples in literature (cf. chapter 4). Although bandwidth seems to be a determining factor for selecting a bus on the first look, it is possible to work around the limitation of certain busses (e.g. to connect imaging sensors) by shifting the heaviest burdens to one or more high-speed busses like USB or Ethernet. Using this is less elegant than a unified bus, but it is a sound and practical approach and hence further considerations are needed to narrow down the selection.

[1] According to private correspondence on 2012-10-25 with Gary Cheng, at the time product manager of DMP's SOC division.

Real-Time Properties

Like most embedded systems also mobile robots need to adhere to at least some real-time deadlines. In our scope propagating the activation of bumpers and cliff sensors within limited time is one obvious example. While the absolute deadlines were not restricted in this project, giving important messages priority is a must.

Multi-master systems such as CAN have to deal with distributed scheduling as well as preemption of transfers unlike busses with a single master such as USB. CAN handles this by pre-configuring priority values of respective messages which are used for arbitration at the start of a transmission (cf. section 3.2). This scheme does only handle concurrently scheduled transactions, there is no preemption of packets already in transfer. Hence even messages with top priority might not be able to be sent immediately. Due to the small payload sizes of CAN this is not a big issue though because the delays are relatively short (about 100 µs to 150 µs for CAN 2.0 at 1 Mbit/s).

Another way to handle multiple masters is to statically schedule (at least some of) the messages. This time-triggered approach is for example done by FlexRay (see section 3.3) or TTEthernet and requires at least some basic time synchronization. Although this greatly enhances predictability, it has also the drawback of increasing the latency of important messages to up to one whole communication cycle. In FlexRay, one cycle at 10 Mbit/s can range from 24 µs ($0.025\,\mu s \times 960\,\mu$Ticks [22, Table B-25]) to 16 000 µs [22, Table B-26], which could be relevant in some use cases (although probably only the most complex applications would come even

Table 5.1: Estimation of the maximum required bandwidth of a backbone bus in a mobile robot (again ordered by bandwidth per sensor).

Sensor Type	Bandwidth per Sensor	Max. Count per Robot	Sum[a]
	[bit/s]		[bit/s]
Bumpers	<1	16	Negligible
Cliff	<1	8	Negligible
Infrared	$\geq$(120 to 400)	16	$\geq$(1920 to 6400)
Ultrasonic	300 to 400	16	$\geq$(4800 to 6400)
GPS	1320	1	1320
AHRS/IMU	6750	1	6750
LIDAR	100×10^3 to 5×10^6	2	200×10^3 to 10×10^6
ToF	40×10^6 to 100×10^6	2[b]	80×10^6 to 200×10^6
RGB-D	$\geq 300 \times 10^6$	2[b]	$\geq 600 \times 10^6$

[a] Or *product*, if you prefer.
[b] Either ToF or RGB-D cameras are used but never both.

close to the maximum).

In single-master systems, like USB, the slaves are polled regularly for new data according to some scheduling algorithm which makes them practically similar to those described above. In USB, devices can communicate their requirements at connection time to the host controller, which then decides if they can be fulfilled (cf. section 3.4, esp. subsection 3.4.5) . For certain high-bandwidth connections poll times of less than 125 µs are possible, but in general the minimum service time is 125 µs for High-speed and 1 ms for Full-speed devices.

Another real-time related issue is synchronization between multiple devices because it allows them to operate at the same time (e.g. measure some value concurrently). Time-triggered architectures like USB or FlexRay usually have some implicit clocks embedded in their protocol to guarantee collision-free bus access. If this clock is available to upper software layers, synchronization of actions is easily possible. Otherwise, like in CAN, a synchronization protocol needs to be implemented on top of the bus, which usually leads to less accuracy and requires additional bandwidth.

Cables, Connectors and Power Supply

Easily manipulable connectors can help to ease the reconfiguration of robots.

CiA 303-1 specifies appropriate pin mappings for using CAN with over a dozen connectors but recommends DE-9 D-subminiature connectors as specified in IEC 60807-3 for general purpose applications, while the CAN specification does not deal with the physical layer at all. These layouts described by CANopen also include a positive supply pin to be connected to a 18 V to 30 V source of undefined power.

The USB 2.0 specification includes a number of connectors with different sizes. All of them include a pin able to supply at least 100 mA at 5 V. Depending on the available power at the upstream port the attached devices can negotiate for up to 500 mA. For industrial use there are also non-standard connectors available (e.g. `http://te.com/products/Industrial-USB-Connectors`).

Maximum cable lengths are not an issue because even the low limit of USB (about 5 m per segment) is sufficient for the envisaged application. Other fieldbusses, which are usually designed to work up to much larger distances, offer their full potential in such small networks. Building customary USB cables by soldering bare connector to a cable is a rather fiddly process unlike working with connectors that are made for crimping or manual soldering (like 8P8C or D-Sub).

Costs, Availability and Other Considerations

The ubiquitousness of USB has led to very low prices of all related components and made them readily availably. This is a very distinctive feature of USB which is especially useful in the area of teaching: Every student could connect the backbone bus of the robot to his/her laptop while developing or for easier debugging.

CAN has also gained some acceptance outside industrial environments but not in the consumer sector and not nearly as much as USB. One can not buy CAN hardware in general con-

No, these are not coffee stains.
Douglas drank tea of course!

sumer electronic shops but only from businesses focusing on automation, robotics or similar, and the pricing is accordingly.

FlexRay on the other hand still is nowhere found but in the automotive sector. The complex protocol is usually implemented in FPGAs only, which makes products expensive and not well suited for smaller robots at all.

Conclusion

The discussion in the past few sections focuses on CAN and USB as transport protocol for sensor data because they are basically the only viable options within requirements. Old fieldbusses like those built on top of TIA-422 (or 485) are often fragmented (i.e. there exist many similar but incompatible, often proprietary competing industry standards without any one of them dominating) and have anachronistic or no high-level protocols at all. The latter is also the reason to rule out I^2C which is very wide-spread and well supported (although often only available for limited bandwidths). Low bandwidth is the major problem with LIN whereas Ethernet is not suitable in any of its variations for example due to complex cabling, poor real-time properties and its low proliferation in microcontrollers.

Choosing between the two main contenders is not easy since they have both advantages and disadvantages relevant to the use case. CAN is somewhat constrained in bandwidth when it comes to transporting the aforementioned more demanding data streams. As explained earlier, this could be mitigated by connecting high-bandwidth devices directly to the host. The protocol itself is somewhat limited too: small payloads and a lack of appropriate scheduling mechanisms which the priority scheme can not compensate for. Power supply and connectors are underspecified as well. CANopen is a step in the right direction, filling many gaps in the original specification and introducing new features such as the OD which could also be used in this project to announce the attached sensors and their parameters to the host.

USB, on the other hand, is very complex and although the scheduling scheme is real-time-capable it is not completely deterministic (unless all details about the host controller and driver are known). Also, the power supply is too weak for many purposes (e.g. driving motors) but appropriate for most sensors (even in groups plus associated microcontroller). Another potentially useful feature, that USB lacks in contrast to CAN, is broadcasting. CAN inherently broadcasts all frames and receivers decide which message IDs they want to subscribe. Due to the proposed architecture where a central ROS instance is responsible to gather and process all sensor readings, this is not an problem. The tree topology used in USB is in general well suited for the task at hand, but the requirement for active hubs creates a burdon for some setups. The standard connectors of USB might be too prone to get loose in the presence of vibrations. The advantages outweigh these issues though: excellent availability, pricing and software support, abundance of bandwidth which often can even be multiplied because of multiple root hubs per host controller, or plug-and-play capabilities. Furthermore, to the best of my knowledge, using USB as a backbone bus for robots has never been tried and documented within academia so far.

Therefore it was decided to use USB.

5.2 Blueprint

This section briefly describes the architecture and main properties of the proposed approach.

As written in the introduction, all higher level processing needs to be calculated within ROS hence all data is to be sent to a computer running a ROS instance where a proxy daemon is responsible to make the information available to the framework. To proactively reduce possible effects of jitter in the transmission and delivery process all data shall be timestamped as early as possible.

To proof that using USB in this context is viable and to evaluate the design a prototype was built. It consists of two micocontroller boards connected to various sensors and a servo motor. The boards use their USB interfaces to transmit the relevant data to a ROS instance running version *Hydro Medusa* on a laptop with Canonical Ubuntu 12.04. The proxy daemon and the high-level communication between it and the daughter boards are described in chapter 8.

The low-level protocol directly on top of USB is an enhanced version of a previously existing remote procedure call (RPC) scheme for microcontrollers developed by the author. Originally conceived to work with Java over a bidirectional connection transmitted via BT, it was ported over to USB for this project and improved to allow microcontrollers to initiate transfers. On the host side the complete protocol software had to be rewritten in C because it was only available in Java before. The implementation of the communication framework is documented in chapter 7.

To allow very early timestamping of messages, a synchronized global clock needs to be established. It exploits the precise period of SOF packets occurrences to synchronize a RTC freely running on all attached microcontrollers. The details of this principle are explained in chapter 6.

Chapter 6

Synchronization

To make computations based on information originating from distributed sensors tolerant to communication jitter, the data can be timestamped at the source. Also, with timestamping any transmission delays become irrelevant as long as there is sufficient time left to react safely because they do not influence the results of calculations. To make this possible at all, the senders need to be synchronized to the host first.

6.1 Theory

The used approach is based on the periodically sent SOFs in USB. As explained in section 3.4 a SOF packet containing an 11 bit long frame ID is sent every 1 ms (or 125 µs for High-speed). Furthermore, all examined USB controllers support raising an interrupt on SOF reception and to read out the current frame ID. The combination of these two properties allows not only to synchronize clocks of USB devices attached to the same bus relatively to each other and the host but also to set the absolute time in a synchronous manner as follows[1].

A local clock (for example implemented by using timer interrupts on a microcontroller) uses the SOF interrupts to adjust its rate under the assumption that the interrupt is recurring regularly. To make the devices also aware of absolute time[2], the host needs to send it in a way so that the devices can associate a SOF interrupt with it to synchronize their local clock accordingly.

6.1.1 Drawbacks

The data clock of USB is not precise but allowed to vary by up to 0.05% (or even 0.25% for non-High-speed devices) [66, 7.1.11]. Likewise, the period of frames (defined by the interval of SOFs) is specified to be 1 ms $\pm$ 500 ns (High-speed: 125 µs $\pm$ 62.5 ns) [66, 7.1.12]. Additionally, each section of a Full-speed hub plus cable is allowed to introduce a delay of 70 ns [66, 7.1.14.1]. Also, it is not guaranteed that a SOF packet is received successfully by all devices at the start

[1] The author does not claim any credit for this idea. It was, for example, discussed in a thread on the Microchip forums before this project even started: `http://www.microchip.com/forums/m329799.aspx`.

[2] Physicists, please pardon me!

of every frame (e.g. because of signal distortion due to interference leading to bit flips and CRC failures).

Because there is no broadcast functionality available in USB, the time references need to be sent to each device individually which costs bandwidth, processing power, buffer space and an additional end point address. Also, associating a frame ID with a real-time reference is also not supported by any (known) operating system and has to be done in user space which reduces accuracy or by changing the kernel.

While these disadvantages are an issue in high-precision applications, they can be tolerated in mobile robots. Alternatively, it is possible to get much better accuracy and precision (in the order of nanoseconds or even below) by using dedicated hardware for the local clocks and for measuring the delays added by hubs and cabling to the communication paths of the individual devices [23].

6.2 Implementation

The implementation consists of two main parts: the daemon (named SyncUSB) running on the host computer that is responsible to send out frame IDs associated with real time, and the synchronization software running on the microcontroller boards. All time values are defined as unsigned 64 bit types representing µs since Unix epoch. Frame numbers are stored in 16 bit types. Communication between the host and devices happens through a single interrupt pipe with a payload size of 10 B (64 + 16 bit) transferring the time first and then the frame ID.

An overview of the implementation containing all major communication and control channels is depicted in Figure 6.1.

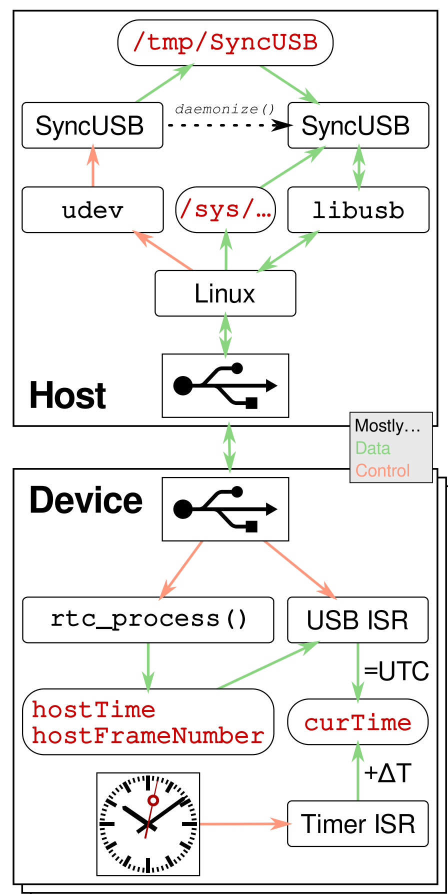


Figure 6.1: Synchronization scheme.

6.2.1 Host Synchronization Daemon

The synchronization daemon gathers time and USB frame information and sends it to all available and compatible devices. It is started by a `udev` rule as shown in Listing 6.1 which matches only compatible devices (i.e. those with appropriate interface attributes including the string descriptor). The daemon is actually not started directly but via the wrapper quoted in Listing 6.2. This has two main advantages: First, the wrapper can be located in a user-editable place which makes changes of parameters easier. Additionally it allows to

Listing 6.1: Udev rule to match synchronization endpoints and to launch SyncUSB.

```
ACTION=="add",\
  SUBSYSTEM=="usb", \
  ATTR{bInterfaceClass}=="ff", \
  ATTR{bInterfaceSubClass}=="34", \
  ATTR{bNumEndpoints}=="01", \
  ATTR{bInterfaceProtocol}=="01", \
  ATTR{bAlternateSetting}==" 0", \
  ATTR{interface}=="SyncUSB", \
  RUN+="udev.sh syncusb"
```

Listing 6.2: Udev launch wrapper (udev.sh).

```
#!/bin/sh

devdir="/sys/$DEVPATH/../"
busnum=$(cat "$devdir/busnum")
devnum=$(cat "$devdir/devnum")
exe=$1
shift
logger -s -i -t "udev.sh" -p daemon.notice "Executing $exe -i 10 -v -b
    $busnum -d $devnum -D $*"
$exe -i 10 -v -b "$busnum" -d "$devnum" -D $*
```

Listing 6.3: (Shortened) usage text of SyncUSB.

```
Usage: syncusb [-d] [-b <bus> -d <dev>] [-D] [-i <secs>] [-u username]

 -b | --busnum <bus>      use device on USB port <bus> (default: any)
 -d | --device <dev>      use USB device with address <dev> (default: any)
 -D | --daemon            daemonize (default: do not)
 -i | --interval <secs>   sleep <secs> seconds between syncs (default: 1)
 -u | --username <user>   if run as root, change to user <user> (default:
      'nobody')
 -v | --verbose           print verbose output (default: do not)
```

easily derive the sysfs directory paths containing the information required to address the right controller and device.

The executable implementing SyncUSB is called syncusb and allows to change some runtime options via the command line arguments shown in Listing 6.3.

One important feature of the daemon is that it is a singleton: only one syncusb is an actual daemon communicating with devices at any time. Only the first instance that successfully creates a specific UNIX domain socket (namely /tmp/SyncUSB) can become responsible. Whenever another one is started it will recognize the existing socket and just inform the running daemon about newly attached devices using exactly the socket that also guarantees the exclusive daemon process. SyncUSB maintains a list of all active compatible devices and provide them with synchronization packets regularly. It detects detached or otherwise non-working devices and removes them from said list automatically. The source code that creates the socket and can be used to distinguish between servers and clients is shown in Listing 6.4.

Platform Aspects

While libusb (or one of its forks) can be used to communicate with USB devices in a relatively platform-independent way, a few important tasks of the daemon require specialized approaches.

A small library (called realtimeify) was written to take advantage of a number of Linux features that improve the real-time capabilities of a process. This includes the prevention of page faults by locking all used memory, disabling any frequency scaling and idle states, using alternative scheduling and shielding the process on a single core (i.e. moving all others away). These operations are Linux-specifc because there exists no standard interface for them (e.g. in POSIX) or some aspects are implementation-defined.

To gather the current frame ID of a USB port the Linux kernel had even to be modified because there was no user-space API for that so far. Internally every Linux host driver provides an implementation of usb_get_current_frame_number() that does mostly what the name suggests[3]. A sysfs interface was added to the kernel that reports the current frame number of each USB controller to user space when the respective file (e.g. /sys/devices/pci0000:00/0000:00:1d.0/usb2/frame_number) is read.

6.2.2 Microcontroller Synchronization Driver

This software package is divided into three major parts: SOF and timer ISRs, and the receiver function rtc_process(). They communicate with each other through these shared variables:

```
1 static volatile time_t hostTime = 0;
2 static volatile int16_t hostFrameNumber = -1;
3 static volatile time_t curTime = 0;
```

[3] It was introduced to implement scheduling of isochronous pipes in a way that is no longer worth pursuing and still using it “is probably a mistake” (Allen Stern on the linux-usb mailing list). Some drivers limit the range of reported values to less than the required 2048 and hence introduce some aliasing.

Listing 6.4: Function to create a singleton daemon with a named UNIX domain socket.

```c
/** Connect or create a singleton socket.
 *  Tries to create a UNIX socket with path \a name and sets \a socket_fd to the
    resulting file descriptor.
 *  \returns
 *    -1 on errors,
 *     0 on successful server bindings,
 *     1 on successful client connects.
 */
int singleton_connect(const char *name, int * const socket_fd) {
  int len, tmpd;
  struct sockaddr_un addr = {0};

  if ((tmpd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) {
    syslog(LOG_CRIT, "Could not create socket: '%s'.", strerror(errno));
    return -1;
  }

  // fill in socket address structure
  addr.sun_family = AF_UNIX;
  strcpy(addr.sun_path, name);
  len = offsetof(struct sockaddr_un, sun_path) + strlen(name);

  int ret;
  unsigned int retries = 1; // at least one to start with abandoned socket
  do {
    // bind the name to the descriptor
    ret = bind(tmpd, (struct sockaddr *)&addr, len);
    *socket_fd = tmpd;
    // if this succeeds there was no daemon before
    if (ret == 0) {
      return 0;
    }
    if (errno == EADDRINUSE) {
      ret = connect(tmpd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un));
      if (ret != 0) {
        if (errno == ECONNREFUSED) {
          syslog(LOG_WARNING, "Could not connect to socket - assuming daemon died.");
          unlink(name);
          continue;
        }
        syslog(LOG_ERR, "Could not connect to socket: '%s'.", strerror(errno));
        continue;
      }
      syslog(LOG_NOTICE, "Daemon is already running.");
      return 1;
    }
    syslog(LOG_ERR, "Could not bind to socket: '%s'.", strerror(errno));
  } while (retries-- > 0);

  syslog(LOG_ERR, "Could neither connect to an existing daemon nor become one.");
  close(tmpd);
  return -1;
}
```

These variables need to be handled with caution (i.e. while interrupts are temporarily disabled) because they can not be accessed atomically and the ISR activations might interrupt a read or write operation in progress, possibly leading to corrupt data.

The receiver function needs to be called periodically to ensure proper handling of packets addressed to the RTC endpoint. Its only purpose is to decode the host time and frame number from received packets and store them in the shared variables.

The SOF ISR is executed on each successfully received SOF and first checks if a new time/frame number pair was received from the host since its last invocation. In that case it resets the timer, and calculates the current absolute time and stores it in `curTime`. The timer ISR on the other hand increments `curTime` by its own period to keep it running between synchronization attempts from the host. Additionally, there exist functions for initialization and to allow other modules access the current time.

6.3 Results

Changing the state of any observable pin on a modern x86 PC takes a while from issuing the command in the CPU until the signal actually reaches the pin. This makes benchmarking the quality of synchronization (especially accuracy) problematic because somehow an external event needs to be triggered to measure and compare the time of its occurrence to those of signals from the synchronized devices.

Only the precision between the synchronized clocks on the microcontroller boards can easily be measured with confidence. A USB logic analyzer was set up to trigger on signal changes of a dedicated pin on each of the two boards. The `rtc_process()` function was extended to toggle those pins whenever a variable containing a timestamp is less than the clock counter, i.e. if and when the trigger time has passed. After an initial synchronization, the trigger time is set equally on both devices to a near point in the future and the logic analyzer is able to record the events. Measurements based on this approach show that a precision of a few µs is achievable.

Chapter 7

J2A — A Control and Data Transfer Protocol for Embedded Systems

The protocol used in the prototype on top of USB was first conceived to control a wearable display via BT. It was originally named *java2arduino (J2A)* because it provided Java code the opportunity to remotely call functions executed on an ArduinoBT. In the course of this thesis the Java part was ported to C and refined to work on USB. Additionally it has been developed further, e.g. to allow devices to initiate communication.

J2A can be divided into two major parts: The client side can use the API to execute functions on the server side and to exchange data with it. The server side implements the API on the microcontroller and allows to easily add user-generated functions that can then be called by the client.

The server part (sometimes referred to a arduino2java (A2J)) is written in C and is designed to be run on Atmel AVR microcontrollers which are used by, among others, the Arduino project, but J2A is in no way limited to Arudino platforms. The protocol can essentially be used on top of any medium that allows to transfer one byte stream in each direction. In the first project the Serial Port Profile (SPP) of BT was used, which emulates a full-duplex TIA-232 connection based on RFCOMM and L2CAP. While there exists a similar scheme for USB (namely CDC), J2A is implemented natively by using two bulk pipes directly.

7.1 Features

The list below briefly describes the most important features of J2A which are explained in detail afterwards.

Distinction of Packets
: The use of a preserved marker at the start of each packet allows to recover from incomplete transmissions.

Sequence Numbers
: To allow upper layers to recognize completely missed frames, each packet contains a

sequence number.

Checksum
A simple checksum algorithm helps to detect transmission errors on unreliable media.

Remote Execution and Data Transfer
The main purpose of the protocol is to allow to easily communicate between an embedded device and a host device. The protocol specifies a standard function prototype (and hence ABI) that the microcontroller software may implement to make certain functionalities available to the host.

Function Name Mapping
Available functions on the server are discoverable by retrieving a string-to-opcode map stored on the server.

Static Properties
Similarly to the above it is possible to store pairs of strings on the devices (e.g. to add unique IDs).

Large Transfers
While the payload of a single packet is limited to 255 B, infrastructure is provided to transfer up to 4 GB by a sliding window scheme.

The first three features mentioned above rely on properties of the J2A packets themselves. Therefore the involved fields of the messages are explained first in the next section in the context of these features.

The general structure of J2A packets is depicted in Figure 7.1.

Start	Sequence#	Opcode	Length	Data	Checksum
1 byte	1 byte	1 byte	1 byte	up to 255 B	1 byte

Figure 7.1: Format of J2A packets (without Byte Stuffing).

7.1.1 Start of Frame and Byte Stuffing

It has to be noted that Figure 7.1 does not necessarily represent what is actually transmitted on the medium: The protocol marks the first byte of every packet with a distinct symbol that is not used anywhere else in the transmission. This allows the receiver to detect the start of any message, even if the previous one got mangled so that it is not known when it ends. To make sure that the marker does not occur elsewhere, the protocol uses an escape sequence similar to that of Point-to-Point Protocol (PPP) (cf. RFC-1549).

Whenever a byte value equaling the start marker is to be sent (in an other byte than the first) the protocol demands that it is escaped by first transmitting an escape character first followed by the original value decremented by one. The receiver will notice the escape character and

deliver the following byte incremented by one upstream. If the escape character itself is to be transmitted as payload, it is escaped in the same way.

The example below shall clarify any dubiety. Let the start marker be `0x12`, the escape character be `0x7D` and

0	1	2	3	4	5	6	7	8
0x12	0x02	0x11	0x04	0x12	0x12	0x13	0x7D	0xD5

the data to be sent, then

0	1	2	3	4	5	6	7	8	9	10	11
0x12	0x02	0x11	0x04	**0x7D**	*0x11*	**0x7D**	*0x11*	0x13	**0x7D**	*0x7C*	0xD5

is the data actually transmitted. The bytes highlighted by a red background in the upper figure need to be escaped. Below, the newly inserted escape characters are **bold**, while the escaped and therefore decremented characters are in *italics*.

7.1.2 Sequence Numbers

Although the majority of underlying protocols do not rearrange bytes (nor whole packets) during transfers, sequence numbers can help the receiver to detect completely missing packets. Therefore, the second byte of each packets contains the value of an 8-bit counter that is incremented with each transmitted packet, resulting in an overflow and repetition of values every 256 packets.

7.1.3 Checksum

A very simple checksum algorithm was added to J2A to make sure it can work even on unreliable medias. It is calculated by adding constants to some fields and XORing all bytes but the first together. The constants to be used are 11 ($0B_h$, 00001011_b) for the Opcode and 97 (61_h, 01100001_b) for the payload length field. This scheme guarantees that the checksum for the most trivial packet (i.e. all zeros) is not 0. While sending a message the checksum calculation is performed before byte stuffing and vice versa at the receiver side. In the byte stuffing example above the checksum is the result of $2 \oplus (11 + 0B) \oplus (4 + 61) \oplus 12 \oplus 12 \oplus 13 \oplus 7D = D5$ (where all numbers are hexadecimal and $\oplus$ represents the XOR operation).

7.1.4 Opcodes for Remote Method Invocation

The actual control flow between server and client can follow two main paths: either from the host to the microcontroller and back, or starting at the microcontroller without an answer sent back (i.e. a so-called server-initiated frame (SIF)).

If the client on the host initiates communication then the third byte of the request sent encodes the function to be executed on the microcontroller. The reply sent back by the microcontroller subsequently contains in the same field the return value of the afore called method or an error code if there was a problem (e.g. bad checksum).

Alternatively, if the server on the microcontroller starts the transmission, the opcode field declares which handler(s) on the host are responsible to process the packet. Multiple handlers

can register themselves for the same opcode. To discriminate the packets sent by the microcontoller, which are either replies to a previous request from the host or a SIF, different markers are used in the first byte (cf. to subsection 7.1.1).

J2A is also prepared to allow clients to probe the server for available functions. When set up correctly, the server will reply with a mapping from function names to opcodes when asked (by calling an ordinary J2A function as well). To bootstrap this process it is mandatory for the client to use the correct opcode to execute the map-fetching function. For this reason opcode 0 is reserved for this purpose and will return an error when called while SIF support is disabled on the server.

7.1.5 Oversize Payloads

Also built on top of the method invocation system is a procedure to transfer amounts of data larger than the payload capacity of a single J2A packet. Due to resource constraints on microcontrollers, only small chunks can be processed at a time. With the help of the scheme dubbed *a2jMany*[1] these chunks can be combined to form a bigger continuous unit eventually (e.g. to write a file to an SD Card). All chunks have to be transmitted sequentially and without overlap.

On the client side J2A can automatically split big payloads and send them to *a2jMany*-compatible methods. It is also possible to reassemble the small chunks in replies and return them as one big array to the caller.

7.2 Implementation

This section describes the implementation of the overall control and communication flow, starting with the architecture of the microcontroller firmware and followed by how user applications on the host can utilize J2A.

7.2.1 Server

An overview of the server architecture is depicted in Figure 7.2.

The ultimate goal of J2A is to make it easy to communicate with user-generated functions on the microcontroller. The function `a2jProcess()` is responsible for scheduling the execution of user methods according to the packets it receives. The packets are stored in some buffers filled by the low-level A2J drivers in cooperation with `a2jTask()`. The latter needs to be executed periodically (possibly by timer interrupts) to handle communication issues not handled by regular ISRs.

At compile time, a jump table (`a2j_jt`) with pointers to all J2A functions is stored in the flash memory of the microcontroller. An example of how this looks like is given in the next section. When `a2jProcess()` reads a valid packet, it uses the function pointer according to the opcode field of the received packet to call the respective method. All methods to be callable

[1] Note the reference to A2J in this name that also equals the function name in the implementation on the microcontroller.

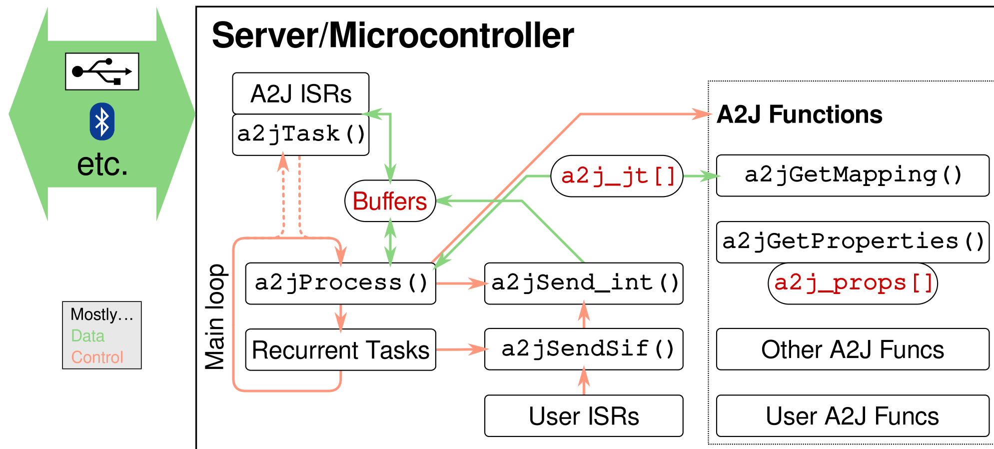


Figure 7.2: Microcontroller side of J2A (dubbed A2J).

by the host have to be referenceable by function pointer type `CMD_P` or `CMD_P_MANY` as shown in Listing 7.1.

Listing 7.1: Function pointer types in A2J.

```
1 typedef uint8_t (*const CMD_P)(uint8_t *const lenp, uint8_t* *const datap);
2 typedef uint8_t (*const CMD_P_MANY)(bool* isLastp, bool isWrite, uint32_t
      *const offsetp, uint8_t *const lenp, uint8_t* *const datap);
```

The first type is useful for applications that do not need to exchange more than 255 B. Respective functions are directly called with a pointer (to a pointer) to the payload (`datap`) and its length (`lenp`) as parameters. Data can easily be sent back by just storing it in the array referenced by the payload parameter (and adapting the length value as needed). It is then automatically transmitted back by `a2jProcess()` after the method returns. The other type is used in the case where more data needs to be transmitted as explained in subsection 7.1.5 and section 7.2.1.

Function Mapping

Because the methods to be executed on behalf of the client are encoded as integer offsets in the jump table while being transferred, a mapping between method names and the corresponding offsets increases the usability of the API. This mapping is optionally stored aside the function pointers in the jump table if enabled (else it saves a bit of flash space). When a client connects, it fetches the mapping with the help of `a2jGetMapping()` on the microcontroller. This allows the user application on the host to call methods just by their names.

The implementation allows to specify a different name for the mapping than is actually used in the source code. The example setup depicted in Listing 7.2 does not use this aliasing but stores all functions with their real name. Listing 7.3 shows what the macros used for the setup actually

do. FUNCMAP just creates a string stored in flash memory and the ADDJT macro is responsible for actually adding the pointer referencing that string to the jump table. The STARTJT macro does not only declare a2j_jt but also adds the pre-defined J2A functions to it and creates appropriate mappings. Note that the definitions rely on the compile time options (e.g. support for the function mapping) and only one instance is shown.

Listing 7.2: Example setup of the jump table and function mapping.

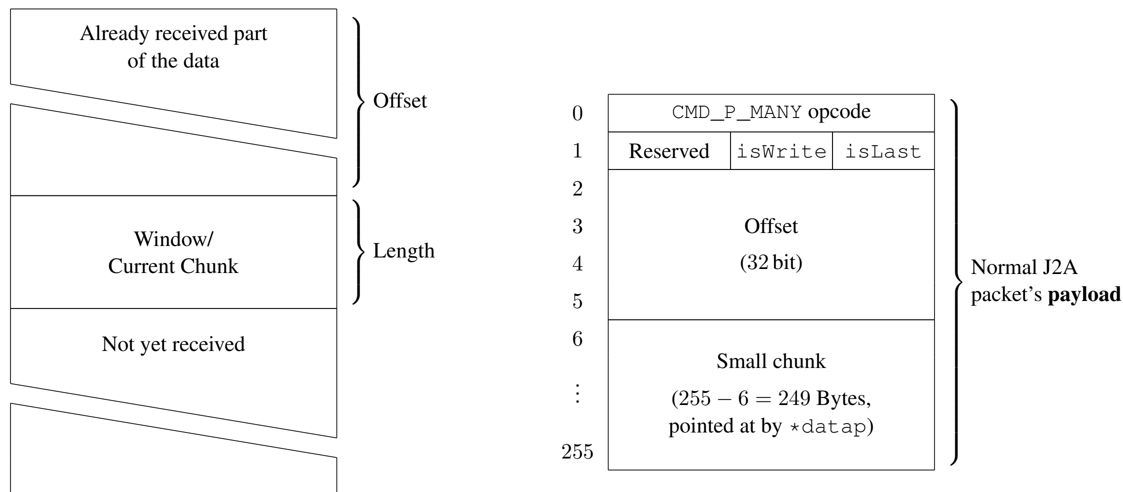
```
/* Function mapping */
FUNCMAP(getTime, getTime)
FUNCMAP(setTrigger, setTrigger)
FUNCMAP(a2j_set_servo, a2j_set_servo)

/* Jump table */
STARTJ
ADDJT(getTime)
ADDJT(setTrigger)
ADDJT(a2j_set_servo)
ENDJT
```

Listing 7.3: Implementation of macros used in the setup of the jump table and function mapping.

```
/** Creates a string \a "alias" in flash accessible by variable \a
    FuncName_map */
#define FUNCMAP(funcName, alias) static const char PROGMEM
    funcName##_map[] = #alias;
/** Appends an entry to the jumptable */
#define ADDJT(funcName) , {&funcName, funcName##_map}
/** Finalizes the jumptable/function mapping */
#define ENDJT }; const uint8_t a2j_jt_elems =
    sizeof(a2j_jt)/sizeof(jt_entry);
/** Start of the jumptable including entries for various (default)
    arduino2j functions.*/
#define STARTJT \
  FUNCMAP(a2jGetMapping, a2jGetMapping) \
  FUNCMAP(a2jMany, a2jMany) \
  FUNCMAP(a2jGetProperties, a2jGetProperties) \
  FUNCMAP(a2jDebug, a2jDebug) \
  FUNCMAP(a2jEcho, a2jEcho) \
  FUNCMAP(a2jEchoMany, a2jEchoMany) \
  const jt_entry PROGMEM a2j_jt[] = { \
  {&a2jGetMapping, a2jGetMapping_map} \
  ADDJT(a2jMany) \
  ADDJT(a2jGetProperties) \
  ADDJT(a2jDebug) \
  ADDJT(a2jEcho)\
  ADDLJT(a2jEchoMany)
```

a2jMany

As explained in subsection 7.1.5, J2A supports transporting large amounts of data by fragmentation. The underlying idea is to transfer the necessary data chunk per chunk in a moving window together with the current offset of the window within the complete data set as illustrated in Bytefield 7.1(a).



(a) Fragmentation scheme of *a2jMany*.

(b) Layout of *a2jMany* payloads.

Bytefield 7.1: Schematics of *a2jMany*.

The scheme is built upon the normal send/receive methods of J2A, that are only able to transmit up to 255 B at once. The sender splits the data into small chunks and sends them to `a2jMany()` which evaluates an additional header at the beginning of the payload as depicted in Bytefield 7.1(b). That header contains the arguments defined by `CMD_P_MANY` (cf. Listing 7.1) as well as the offset of the `CMD_P_MANY`-compatible function in the jump table (NB: the original J2A opcode field is already used to *call* `a2jMany()`). It is follow by the actual payload of up to 249 B.

Most parameters are declared as pointers so that callees can also assign new values for two-way communication (just as it is done in the `CMD_P` case). To indicate that the big chunk is complete, `isLastp` is set to true in the last small chunk to be transferred.

Properties

It may be convenient to store properties (e.g. individual characteristics or serial numbers) on devices and be able to read them out from host. J2A supports this by making key-value-pairs stored in the flash available to clients via `a2jGetProperties()` which is using the *a2jMany* scheme described above to allow a total of up to 4 GB of properties. Listing 7.4 shows the implementation of the associated macros and their use. Note the resemblance to the jump table used for scheduling and in function mapping.

Listing 7.4: Implementation and example use of J2A properties.

```
1  /** Header for the properties. Needs to be called first.*/
2  #define STARTPROPS unsigned char const PROGMEM a2j_props[] = {
3  /** Adds a mapping from \a key to \a value. */
4  #define ADDPROP(key,value)  #key "\0"  #value "\0"
5  #define STRFY(x) #x
6  /** Adds a mapping from \a key to the stringified version of \a value. */
7  #define ADDPROPEXP(key,value) #key "\0" STRFY(value) "\0"
8  /** Finalizes the properties. */
9  #define ENDPROPS }; const uint8_t a2j_props_size = sizeof(a2j_props) - 1;
10
11 /* Example */
12 #define MACRO 13
13 STARTPROPS
14 ADDPROP(serial, 0x012F0B04) /* "serial" -> "0x012F0B04" */
15 ADDPROPEXP(MACRO, MACRO) /* "MACRO" -> "13" */
16 ENDPROPS
```

7.2.2 Client

This section describes the architecture of an A2J application on the host using the C implementation. Any such program contains at least two threads as shown in Figure 7.3.

The main thread executes the user application and issues communication requests via J2A functions. Depending on the implementation of the underlying protocol, another thread might be required to handle transmissions. One such example is the library used for accessing USB, namely *libusb*, that requires at least periodic calls to an event handler. In the current J2A implementation a dedicated thread is used for that purpose and another one to process replies and execute SIF handlers respectively (cf. next section).

All communication between the threads relies on `struct j2a_handle` that contains pointers to all data structures related to one connection such as caches of function mappings and properties, mutexes, condition variables, buffers, handlers etc. A pointer to this struct has to be passed to all J2A functions and is obtained by the user application as return value from `j2a_connect()`.

Server-initiated Frames

For the use case at hand, it does not make sense to let the host probe all daughter boards for new data all the time. Instead the applications running on the microcontrollers should be able to initiate a transfer when new data is available. This is made possible by using a different marker for the start of the respective packets which is then used by the receiver thread on the host to distinguish it from reply packets. Processing of normal replies is left to the main thread while SIF handlers are run within the receiver thread. At the reception of a SIF the thread calls the handlers that were registered for the respective opcode of the packet. Multiple prospective handlers can be registered for individual opcodes as well as a single SIF handler for multiple opcodes.

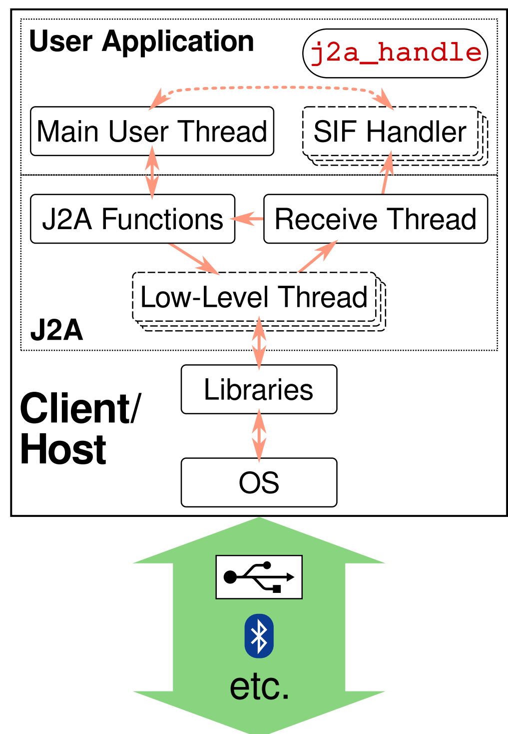


Figure 7.3: Host side of J2A.

Chapter 8

Host Software Interface

All attached sensors have to be defined in a computer-readable manner and the host needs to be able to query this definition from the daughter boards.

This allows to feed data into the ROS framework without requiring much intervention by the end user who wants to utilize the data and design subscribing ROS nodes. The work to develop sensor drivers and to integrate multiple drivers into a workable firmware is also reduced.

This chapter describes how this is achieved in the prototype by explaining its inner workings, starting with the unified interface to sensor drivers on the microcontroller. Based on that, the integration into ROS will be clarified in the second part. An example of an actual workflow execution that shows the ease of integrating a newly attached sensor is illustrated in chapter 9.

When using the prototype as a ROS developer only two statements are required to initialize the framework. Additionally, to build a workable *subscriber* one needs only to know some of the data specified in the transducer descriptors (e.g. to infer the correct *topic* in ROS) and the semantics of the incoming data. A firmware developer, on the other hand, is provided with a straightforward communication interface to transmit new sensor values that is generated automatically from the descriptors in the microcontroller firmware.

Figure 8.1 provides a quick overview of the whole system with the most important components for this chapter in bold.

8.1 Transducer Definitions and Firmware Interface

An elaborate framework of C macros has been designed using X-macros[1] to ease adding or modifying the transducer configuration of a single board. It takes information provided by the developer in a single, plainly formatted file and generates boilerplate C code to interface the hardware drivers with J2A. An example of the contents of such a definition file is given in the next chapter in Listing 9.2. The meaning of the fields are briefly described below.

Name

The first parameter is used as the topic name in ROS that clients can subscribe to receive updates.

[1] https://en.wikibooks.org/wiki/C_Programming/Preprocessor#X-Macros

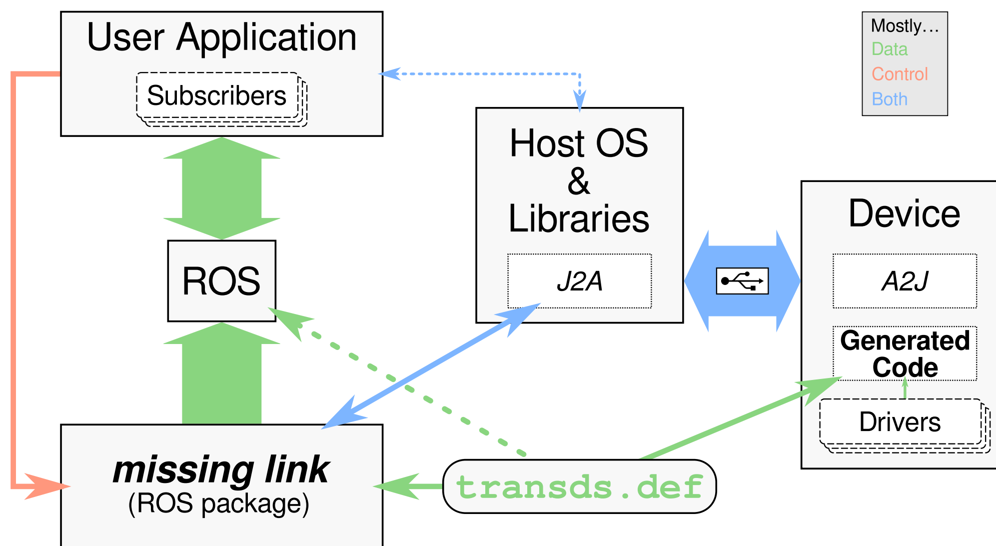


Figure 8.1: Overview of the prototype system.

ID
: The ID has to be unique per microcontroller and is primarily important because it is used as opcode in SIF packets.

Prefix
: The generated code relies a lot on the third field to discriminate entities associated with the distinct driver modules.

Model
: To allow the receivers to interpret the data correctly they usually have to know the sensor model.

Number of Channels
: Each driver can provide data of multiple channels (as long as the data format is equal).

Data Type
: This refers to the C data type of a single measurement value. It is used a lot in code generation, not only on the microcontroller, but also the host.

N/A
: The last field describes the value to communicate illegal or unchanged measurement results.

All driver modules have to strictly follow a few rules to allow the automatic code generation to work. Alternatively, it might be possible to write wrappers around existing driver code but this is not further considered. For details please refer to section 9.1.

8.1.1 Firmware Code Generation

X-macros are used differently to regular C macros. Instead of defining the implementation of the macro body once and *calling* it multiple times with different arguments, the parameters for X-macros are written only once and saved in a separate file. To actually generate code with them, their body is defined at multiple places just before the file is `#include`d. This allows to generate far more complex programs because the created code of multiple *calls* is no longer required to be self-contained but can be interweaved. The second important advantage is that the produced instructions are not written where the macro user places them but where the developer of the macro completely controls the environment. This frees the user from having to know the details of the macro implementation because their only interaction with them is to define their parameters. Regular macros on the other hand usuallly require to embed their generated code more directly.

Listing 8.1 shows one example of how the `SENSORS` definitions supplied by the user are exploited by the X-macros to generate the declaration of the callback registration function explained in the previous section. It has to be noted that this example on its own could easily be solved by a regular macros.

Not every instantiation of an X-macro is required to use all parameters. More often they will only require a subset of those supplied. In the given example the only two used fields are highlighted.

Note that for every entry in the definition file, exactly one function declaration is created by this scheme. After each step the macro needs to be `#undef`ed to be reused as shown.

Listing 8.1: One instance of the use of X-macros in code generation.

```
1 // Declare driver functions (spares us from including any headers)
2 #define SENSOR(name, id, prefix, model, chan_count, data_type, na) \
3 void prefix ## _set_cb(void(*new_value_cb)(uint8_t chan, data_type val));
4 #include "transds.def"
5 #undef SENSOR
```

The code generator declares, defines and manages all needed variables (e.g. for buffers and mutexes). It also takes care of the actual implementation of the callback, which gathers timestamps and correlates them with each sensor value. These pairs of time and sensor values are stored in a buffer together with a dirty flag (one per each driver). A processing task which has to be called periodically checks these flags and sends the buffer contents to the host in case of new data.

Additionally, the macros create a static length check to make sure that sensor updates can be sent in a single J2A frame, store the transducer definitions in flash memory and hence make them accessible by a single J2A function (i.e. `a2jGetTransds()`).

Listing 8.1 shows how the generated code interacts with the rest of the firmware.

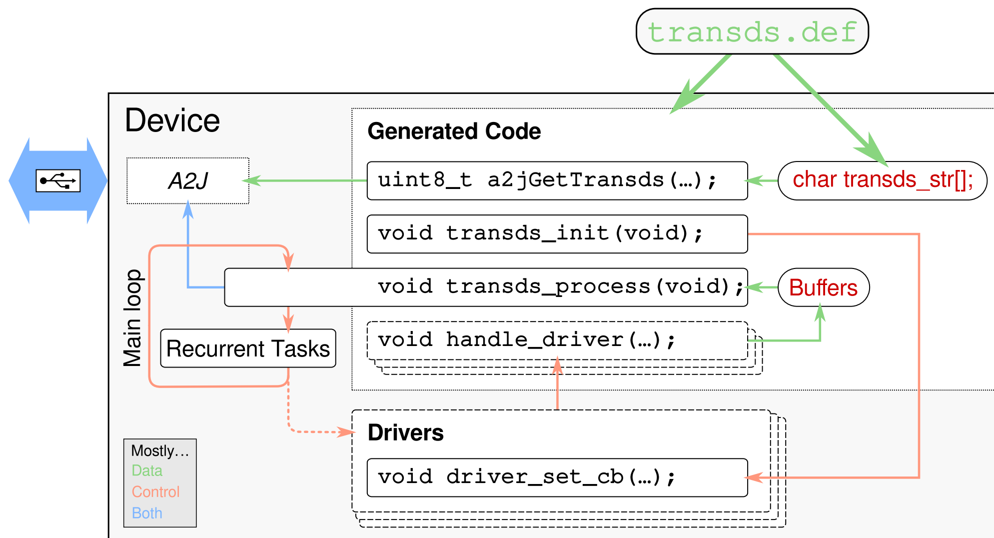


Figure 8.2: Overview of the firmware architecture based on the generated code.

8.2 ROS Integration

The communication system of ROS is based on *publishers* that post messages to *topics* and *subscribers* that receive them. The message types need to be defined in dedicated files (one type per file) at compile time and are used to generate stubs by the ROS build system (which is based on CMake).

Because C++ does neither support reflection nor dynamic class loading it is not possible to interact with the ROS messaging system completely dynamically. The built prototype has nevertheless achieved a very loose coupling that allows for easy integration of new messages.

The J2A function `a2jGetTransds()`, briefly mentioned in the last paragraph of the previous section, forms a bridge to ROS as follows. It allows the host to retrieve sensor configuration including the expected data types and amounts from any attached board. This is used twofold in the prototype.

8.2.1 ROS Messages

First, a small C application (named—behold the creativity—*create_msgs*) queries all attached boards and writes out ROS message files corresponding to the transducer descriptors and the known behavior of the generated code explained in section 8.1. The written file is named after the `name` field in the definition. As an example the result for the quadrature encoder that is used throughout this section is shown in Listing 8.2 below. In each line the type of the field is given on the left hand while the name is written at the end. The first row of all generated files is equal and indicates a variable-sized array of ROS' own time representation type. In the second line a second array is defined whose type is depending on the data type in the descriptor. Note that ROS uses their own naming for integer types which is missing the '_t' suffix used in C's

Listing 8.2: Content of quad.msg.

```
time[] times
int16[] values
```

stdint.h.

The resulting file(s) need to be made available to ROS' build system. Normally this is done by specifying each file manually in the project's CMakeLists.txt but by exploiting CMake's scriptability this can be automated as depicted in Listing 8.3.

Listing 8.3: CMake script to include all *.msg files in a ROS project.

```
file(GLOB _msg_paths "${CMAKE_CURRENT_SOURCE_DIR}/msg/*.msg")
foreach(_path ${_msg_paths})
  get_filename_component(_path ${_msg_paths} NAME)
  add_message_files(FILES ${_path})
endforeach(${_path})
```

The second and less trivial environment where the transducer definitions are used is explained in the following section.

8.2.2 Missing Link

The most important pieces of the developed framework within ROS are illustrated in Figure 8.3. It solely relies on J2A for communication with the microcontrollers (depicted on the top left). Beside normal ROS setup a node is only required to do very little configuration to initialize the framework as shown in Listing 8.4. To understand theses lines one needs to grasp the separation of classes in the framework

Listing 8.4: Setting up *missing_link*.

```
  missing_link::transducer_factory::init();
  missing_link::transducers &t = missing_link::transducers::getInstance();
```

The separation is a result of the static type system of C++ which for example does not allow to put instances of template classes into a std::vector unless the type parameters match exactly. To solve the concrete problem just mentioned, an abstract class transducerI was written that is not templated and can be used instead of concrete implementations (i.e. transducer objects) where it is needed. Additionally, the transducerI interface implements the static method handle_sif_packet_static() which is required to let the plain C code in J2A call back into missing_link. This is only possible with static methods because plain C is not able to use pointers to C++ instances correctly which would be needed to call instance methods. The static method within C++ however is then able to call the correct instance method by casting an opaque pointer given as argument to a reference to the respective instance (cf. Listing 8.5).

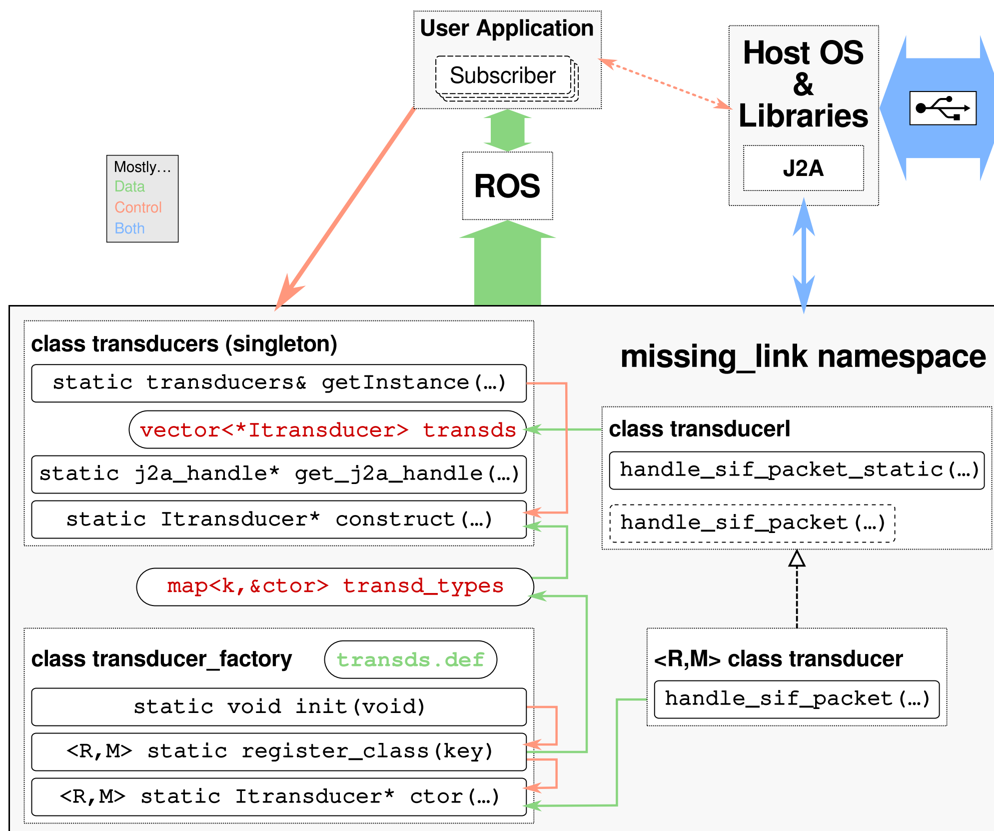


Figure 8.3: Overview of the proxy interface for ROS (i.e. *missing_link*).

The second function shown in that figure is implemented in the subclass `transducer`. It lists the unified J2A SIF handler remotely 'called' by the generated code on the microcontroller in accordance with the descriptor. It is responsible for converting the binary data from the microcontroller into ROS messages.

The listing also highlights the two type parameters used for the template class. `MSG_T` represents the message type of the respective sensor as created by *create_msgs* and `RAW_T` is the C data type of measurements. The use of templates allows to write this code only once and instantiate it with any type required. Also note that the timestamp from the measurements are scaled in this central location to match the time representation of ROS.

As can bee seen in Figure 8.3 instances of `transducer` are stored in member variable `transds` of the singleton `transducers` but how are they created? The answer to this question also explains most of the complexity of the left half of the figure. To spare the user from instancing transducers for every message type in the system manually, *missing_link* does this automatically.

The factory class `transducer_factory` does not produce `transducer` objects directly but function pointers to the respective constructors and stores them in a `map` with the name of the transducer as key. These mappings are generated by the `init()` function with

Listing 8.5: Implementation of the message type-agnostic SIF handlers.

```
void transducerI::handle_sif_packet_static(struct j2a_sif_packet *sif) {
  if (sif->user_data == NULL)
    return;

   static_cast<transducerI *>(sif->user_data)->handle_sif_packet(sif);
}
...
void handle_sif_packet(struct j2a_sif_packet *sif) {
  j2a_packet *p = &sif->p;
  MSG_T msg;
  msg.times.reserve(chan_count);
  msg.values.reserve(chan_count);
  for (int i = 0; i < chan_count; i++) {
    uint64_t j2a_time = fromArray(uint64_t, p->msg, sizeof(RAW_T) * i +
        sizeof(uint64_t) * i);
    msg.times.push_back(ros::Time(j2a_time / US_PER_SEC, (j2a_time %
        US_PER_SEC) * 1000));
    msg.values.push_back(fromArray(RAW_T, p->msg, sizeof(RAW_T) * i +
        sizeof(uint64_t) * (i + 1)));
  }
  pub.publish(msg);
}
```

the help of X-macros like the code generator for the microcontroller as depicted in Listing 8.6. Unlike previously where a definition file was responsible for one board only the file imported here has to accommodate information on all boards. This is easily accomplished by leveraging standard `#include` statements. The important point about `init()` is that it is not templated and hence can be called by the user without detailed knowledge of the message types (cf. Listing 8.4).

The second method, that the user application needs to call to initialize *missing_link*, is `getInstance()` of the `transducers` class which creates a singleton that automatically cleans up before the application ends by exploiting the Resource Acquisition Is Initialization (RAII) principle. The first call that creates the singleton also sets up the core parts of `missing_link`: It initializes J2A and connects all boards, fetches and parses the transducer description and instantiates `transducer` objects accordingly with the help function pointers stored by the factory class.

Furthermore, if a user program has to communicate via J2A directly, it can acquire the `j2a_handle` matching a transducer via `transducers::get_j2a_handle()`.

Listing 8.6: Definition of transducer_factory.

```
1  class transducer_factory {
2  public:
3  static void init(void) {
4  #define SENSOR(iname, iid, iprefix, imodel, ichan_count, idata_type, ina) \
5    register_class<idata_type, the_missing_link::iname>(std::string(#iname));
6  #include "transds.def"
7  #undef SENSOR
8  }
9
10 private:
11 template <typename RAW_T, class MSG_T>
12 static transducerI *constructor(j2a_handle *comm, std::vector<std::string>
       args) {
13   return new transducer<RAW_T, MSG_T>(comm, args);
14 }
15
16 template <typename RAW_T, class MSG_T>
17 static void register_class(std::string key) {
18   transducers::transd_types.insert(std::make_pair(key, &constructor<RAW_T,
         MSG_T>));
19 }
20 }; // class transducer_factory
```

Chapter 9

Results

The last three chapters have dealt with the implementation details of the prototype system. To clarify the benefits to the user, a complete execution of the workflow needed to build a new sensor board and to integrate it into a ROS setup is explained. It begins with the design of drivers for the peripheral modules of the microcontroller interfacing the sensors in question. The next section deals with connecting these drivers with the generated code of the system while section 9.2 instructs on building a ROS application using the infrastructure provided by the *missing link* framework. At the end a practical test application is described.

9.1 Creation of Sensor Board Firmware

The generated code that is used to transmit sensor values to the host via A2J, as explained previously, requires to be informed of new measurement values via a callback. The module for the quadrature encoder implemented in the prototype, provides the function declared in below and executes the given callback from its pin change ISRs.

```
1 void quad_set_cb(void(*new_value_func)(uint8_t channel, int16_t value));
```

The drivers are free to report new values at any rate, but since there is no queue buffer not all values might be transferred to the host but get overwritten by too fast updates. The driver code must provide an interface for the generated code to set (or add) a function pointer that is called by the driver when new data becomes available. It has to be in the form given in Listing 9.1 which is explained next.

Listing 9.1: Interface to configure a callback for sensor updates.

```
1 void PREFIX_set_cb(void(*new_value_func)(uint8_t channel, TYPE value));
```

The two parameters of the callback give the channel number (to support up to 256 equal sensors on one board) and the new value respectively. The latter's type depends on the sensor and can be chosen by the driver developer. Currently only plain C data types are supported, but that is no fundamental limit and more complex data types could be implemented.

Multiple drivers per device are supported. To distinguish them, the functions to configure the callback need to be named differently from each other. This is accomplished with the prefix

mentioned in the declaration above. This and other details of a sensor module (e.g. the data type) need to be provided to the build process by the firmware programmer in a file called `transds.def`. The contents of one such file are shown in Listing 9.2. A description of all fields was given in section 8.1. All data supplied in these files are stored in flash and are readable by the host later.

Listing 9.2: Example definitions to be stored in `transds.def`

```
1 /* Name, ID, Prefix, Model, Number of Channels, Data Type, Value
      indicating N/A */
2 SENSOR(quad, 0, quad, HEDS-6500, 2, int16_t, -1) \
3 SENSOR(dist_ir, 1, sharp, GP2Y0A02YK, 8, uint16_t, 0) \
4 SENSOR(dist_us, 2, srf02, SRF02, 12, uint32_t, 0x00FFFFFF)
```

9.1.1 Initialization and Processing

At startup the callbacks need to be initialized. To do so the microcontroller program has to execute `transds_init`. Besides that the firmware needs also to call `transds_process()` regularly. This function is responsible for all communication handling and sends data supplied via the callbacks to the host. The function declarations are available via `transds.h` and are listed in Listing 9.3

Listing 9.3: Functions exported by `transds.h`

```
1 void transds_init(void);
2 void transds_process(void);
```

9.2 Integration with ROS/`the_missing_link`

All data sent by microcontrollers on the daughter boards is automatically made available by the ROS package named `the_missing_link` (NB: The C++ namespace used within the package is `missing_link` only).

9.2.1 Preparation

The messages in ROS are strongly typed and are required to be known at compile time. The necessary information is provided to the build system by files (usually with a `.msg` suffix). Their exact layout and meaning is described in subsection 8.2.1.

Since the necessary information to create them is stored in the daughter boards and is available to the host, they can be generated automatically. A small C application (named—behold the creativity—*create_msgs*) queries all attached boards and writes out ROS message files corresponding to the transducer descriptors. The messages used by *missing link* are named after the

name field specified in the transducer descriptors explained in the previous section. In Listing 9.4 the output of a `create_msgs` run is displayed.

Listing 9.4: Execution of create_msgs.

```
$ ./create_msgs
1 transducer(s) decoded correctly from device 0.
name=quad, id=0, prefix=quad, model=HEDS-6500, chan_count=2,
    data_type=int16_t, N/A=-1
$
[...]
```

The resulting files currently need to be copied manually to the `msg` directory of the *missing link* project and are automatically picked up when compiling is done with `catkin_make`. The compilation process does also require the `transds.def` file used to build the microcontroller firmware. This could also be generated automatically like the message files but is not yet implemented. If messages from more than one board are used then either their data needs to be copied into a single file or a new file includes the existing files like shown in Listing 9.5.

Listing 9.5: How to combine multiple `transds.def` files

```
1 #include "../../firmware1/transds.def"
2 #include "../../firmware2/transds.def"
```

9.2.2 Application

The missing link module is started automatically when a singleton of the `transducers` class is acquired by calling `transducers::getInstance()`. Before that the `transducer_-factory` needs to be initialized. Because this requires the inclusion of header files generated from the `.msg` files as shown in Listing 9.6, this is not fully automated yet. The library is deinitialized automatically when all references obtained by `getInstance()` goes out of scope. As long as the library is active, data received from the microcontrollers is published automatically to their respective topics.

Listing 9.6: Setting up `missing_link`.

```
1 // Message types used in ROS communication need to be included before
    trancducers.hpp
2 #include "the_missing_link/quad.h"
3 #include "transducers.hpp"
4 [...]
5 missing_link::transducer_factory::init();
6 missing_link::transducers &t = missing_link::transducers::getInstance();
```

To make use of them the user application needs to subscribe to topics of interest. An example of how this is accomplished is given in Listing 9.7. The resulting output is shown in Listing 9.8.

Listing 9.7: Example Subscriber

```
1 void demo_quad_cb(const the_missing_link::quad::ConstPtr &msg) {
2   std::string time = transducerI::format_timeval(&(msg->times[0]));
3   uint16_t quad = msg->values[0];
4   ROS_INFO("Received quad value %d from %s", quad, time.c_str());
5 }
6 [...]
7   ros::NodeHandle n;
8   ros::Subscriber sub = n.subscribe("quad", 1000, demo_quad_cb);
```

Listing 9.8: Example Output

```
$ roscore &
[...]
$ ./demo_pwm
[...]
[ INFO] [1398151882.449752093]: Received quad value 1419 from 2014-04-22
    09:31:22.436640000
[ INFO] [1398151882.460496204]: Received quad value 1418 from 2014-04-22
    09:31:22.438448000
[...]
```

9.2.3 Timing

As can be seen in Listing 9.8 the delays between data acquisition on the microcontroller and the reception in the ROS subscribed is quite big. In initial tests they were found to be in the range of 10 ms to 25 ms of which about 10 ms are due to latencies internal to ROS (when compared to a plain C implementation). The problem may stem from a bug in the microcontroller or host program, for example within the time synchronization module on the daughter boards which is responsible to deliver the timestamps in the first place, or there might be a timing issue in the synchronization/coordination of the ISRs. Additional testing is required to find the underlying cause.

9.3 Test Application

The concept was tested on real hardware with two Atmel-based microcontroller boards. Both boards are bus-powered and host an Atmel 8 bit microcontroller running at 16 MHz with 32 kB of flash memory each. One of them is an Arduino Micro with an ATmega32U4 with 2.5 kB of RAM. The other board is a custom-made prototyping board featuring an ATmega32U2 with only 1 kB RAM.

Besides the implementation of the J2A protocol and the generated code described in subsection 8.1.1, the firmware consists of various mostly self-developed drivers. The largest exception is the excellent USB library called Lightweight USB Framework for AVRs (LUFA) created and maintained by Dean Camera.

To practically test the prototype a Hitec HS-645MG servo is controlled indirectly by an Avago Technologies HEDS-6500 quadruple encoder. The sensor data is read in by the host, converted to a proportional deflection value and sent back to the board which sets up the PWM for the servo accordingly.

Figure 9.1 shows a picture of the whole setup.

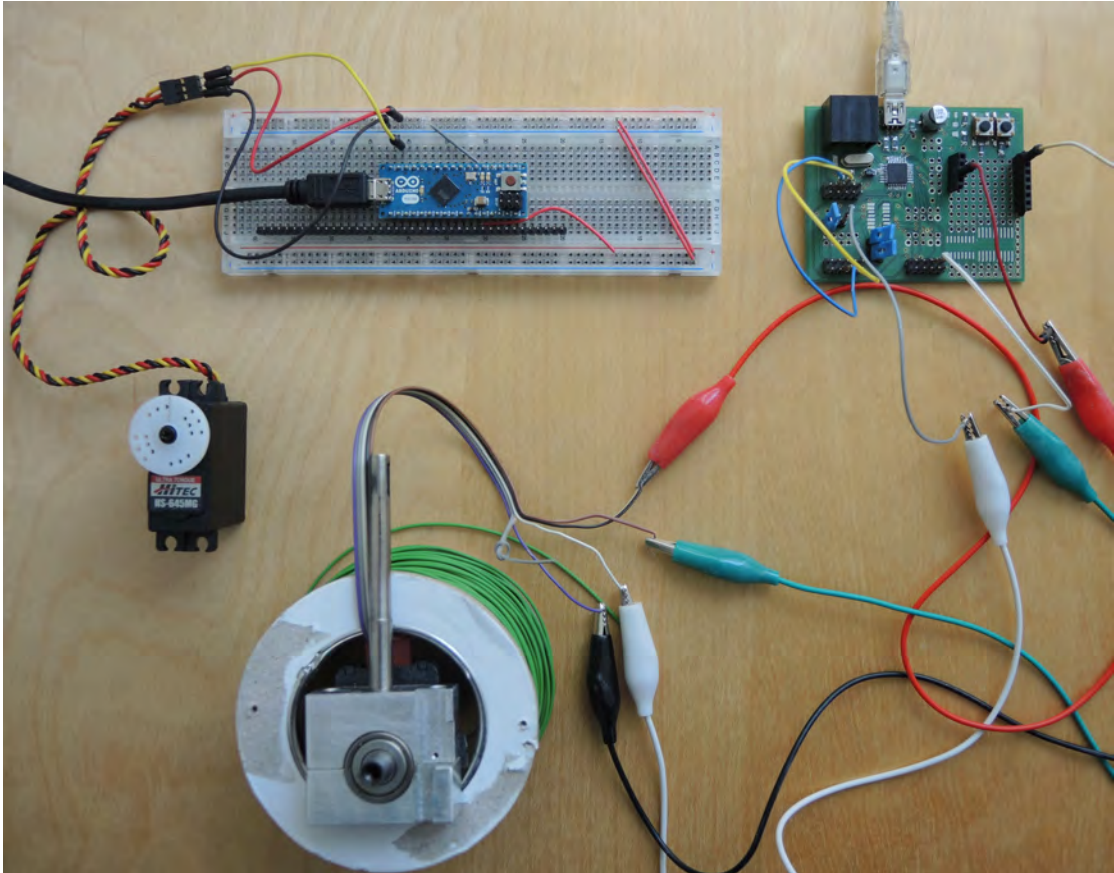


Figure 9.1: Test setup. Arduino in the middle on the top, ATmega32U2 next to it on the right, Hitec Servo on the left and quadruple encoder mounted inside a metal casing on the bottom.

Chapter 10

Conclusion

This chapter briefly sums up the contributions of the thesis and states possible enhancements.

10.1 Summary

The aim of this thesis was to investigate possibilities to mitigate the problems created by the use of multiple busses in mobile robots with a focus on systems that can be used in research as well as teaching. After providing an overview about the working principles of transducers used in the field of mobile robots in chapter 2 a survey of various (field) busses in chapter 3 was given. In chapter 4 related work in academia as well as in commercial environments was investigated.

The second part of the thesis, which focuses on the implemented approach, begins with chapter 5. It describes the given requirements, resulting problems and their possible solutions in detail and lays out the work to be done.

The synchronization scheme discussed in chapter 6 proved to be successful. With the wisdom of hindsight its addition to the prototype was quite fortunate as tests showed the latency between sensor reads and the reception in host programs to be about 10 ms to 25 ms — a non-negligible amount in some applications. The developed implementation is self-contained and might be useful for other projects that require a distributed clock in embedded systems at a low cost.

Likewise, the communication protocol J2A (cf. chapter 7) was designed with other use cases in mind and is therefore reusable. Possible applications include any task that demands for a communication channel between an embedded controller and a full-blown host computer.

The proxy to provide independent ROS nodes with streams of sensor data is described in chapter 8. It minimizes the efforts of hooking up new sensor configuration by providing a unified interface for microcontroller drivers and generating boilerplate code required for communication on the host as well the microcontroller. The description of a setup is solely done by tiny text files specifying the attached transducers.

In summary, a workable prototype was built which demonstrates possible ways to achieve a unification of transducer access in mobile robots. It also proves that USB can successfully be used in this field.

10.2 Future Prospects

Like any other project, the prototype described herein, is not perfect and there is still much room for improvement.

Additional testing is required to find the cause of the yet unexplainable delays seen in transmissions with the prototype.

The synchronization scheme relies on the availability of operating system services that allow to correlate precise timestamps with current USB frames. The implementation of the prototype uses a modified Linux kernel to accomplish this in a rather limited approach. With some effort it should be possible to work out a set of changes that is acceptable for upstream kernel developers.

Currently the algorithm to synchronize the distributed clocks abruptly resets the local clocks when it receives an anticipated SOF packet. This can lead to readouts that seem to move backwards in time which is obviously problematic. A continuous resynchronization that speeds up or slows down the local clock according to the calculated offset would solve this issue.

The sensor abstraction is rather crude and lacking features. Direct support for actuators is completely missing and there is no way to disable or otherwise configure attached (sets of) sensors. While this can easily be implemented on top of J2A (as has been done in the prototype already) it makes sense to unify approaches early to exploit synergies. The local clocks could be used to synchronize measurements or activations of actuators.

Some values in the transducer descriptor are not mandatory to be compile time constants. For example, it might be useful to allow the user to change the topic name at run time.

USB offers abundant bandwidth compared to other current field busses. It would be interesting to stress test single boards as well as whole systems to investigate performance and discover deficiencies.

The current approach is focused on Atmel microcontrollers but design and implementation of all components abstract the underlying hardware away as much as possible so that it should be easy to port it over to other architectures. Implementing J2A and related parts on other embedded platforms could confirm this assumption.

Furthermore, the build process on the host could be further simplified by automatic generation of the transducer descriptors obtained from the devices and by generating respective `#include` directives to initialize `transducer_factory` without including the needed message headers manually.

Bibliography

[1] Allegro MicroSystems Inc., Worcester, Massachusetts 01615, U.S.A. *A3967. Microstepping Driver with Translator*, 2003. Datasheet. URL: http://www.allegromicro.com/~/media/Files/Datasheets/A3967-Datasheet.ashx. Cited on page 14.

[2] R. Anderson, H. R. Bilger, and G. E. Stedman. "Sagnac" effect: A century of Earth-rotated interferometers. *American Journal of Physics*, 62(11):975–985, 1994. doi:10.1119/1.17656. Cited on page 9.

[3] Mario N. Armenise, C. Ciminelli, F. Dell'Olio, and V. M. N. Passaro. *Advances in Gyroscope Technologies*. Springer, 2011. doi:10.1007/978-3-642-15494-2. Cited on page 10.

[4] Atmel Corp., San José, CA 95131 U.S.A. *AVR1600: Using the XMEGA Quadrature Decoder*, 2008. Application Note. URL: http://www.atmel.com/Images/doc8109.pdf. Cited on page 8.

[5] Avago Technologies Limited, San José, CA 95131 U.S.A. *HCTL-2032, HCTL-2032-SC, HCTL-2032-SCT, HCTL-2022*, 2007. Datasheet. URL: http://www.avagotech.com/docs/AV02-0096EN. Cited on page 8.

[6] Jan Axelson. *USB Complete: The Developer's Guide*. Lakeview Research, 4th edition, 2009. Cited on page 26.

[7] O. Barker, R. Beranek, and M. Ahmadi. Design of a 13 degree-of-freedom biped robot with a CAN-based distributed digital control system. In *International Conference on Advanced Intelligent Mechatronics (AIM) 2010*, pages 836–841. IEEE, July 2010. doi:10.1109/AIM.2010.5695770. Cited on page 35.

[8] R. Belschner, J. Berwanger, F. Bogenberger, C. Ebner, H. Eisele, B. Elend, T.M. Forest, T. Führer, P. Fuhrmann, F. Hartwich, et al. FlexRay communication protocol, October 2003. EP1355456. URL: http://register.epo.org/application?number=EP02008171. Cited on page 25.

[9] Johann Borenstein, Hobart R. Everett, and Liqiang Feng. *Where am I? Sensors and methods for mobile robot positioning*. 1996. URL: http://www-personal.umich.edu/~johannb/Papers/pos96rep.pdf. Cited on page 8.

[10] Henk Boterenbrood. CANopen high-level protocol for CAN-bus. Technical report, March 2000. Cited on page 23.

[11] Kévin Bruget, Benoît Clement, Olivier Reynet, and Bernt Weber. An Arduino Compatible CAN Bus Architecture for Sailing Applications. In *International Robotic Sailing Conference (IRSC) 2013*, pages 37–50. Springer International Publishing, 2014. doi:10.1007/978-3-319-02276-5_4. Cited on page 35.

[12] Thomas Bräunl. *Embedded robotics: mobile robot design and applications with embedded systems*. Springer, 2nd edition, 2008. Cited on page 13.

[13] RobertI. Davis, Alan Burns, ReinderJ. Bril, and JohanJ. Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007. doi:10.1007/s11241-007-9012-7. Cited on page 21.

[14] Safaa Dawoud. GNSS principles and comparison. 2012. URL: http://www.snet.tu-berlin.de/fileadmin/fg220/courses/WS1112/snet-project/gnss-principles-and-comparison_dawoud.pdf. Cited on page 8.

[15] A. Depari, P. Ferrari, A. Flammini, D. Marioli, E. Sisinni, and A. Taroni. IEEE1451 smart sensors supporting USB connectivity. In *Proceedings of the ISA/IEEE Sensors for Industry Conference, 2004.*, pages 177–182, 2004. doi:10.1109/SFICON.2004.1287156. Cited on page 34.

[16] A. Depari, A. Flammini, D. Marioli, and A. Taroni. USB sensor network for industrial applications. In *Proceedings of the 21st IEEE Instrumentation and Measurement Technology Conference, 2004. IMTC 04.*, volume 2, pages 1203–1207 Vol.2, May 2004. doi:10.1109/IMTC.2004.1351280. Cited on page 34.

[17] DMP Electronics Inc., New Taipei City 24890, Taiwan. *RoBoard RB-110 Manual*. Datasheet. URL: http://www.roboard.com/Files/RB-110/RoBoard_RB-110_v1r1C.pdf. Cited on page 40.

[18] C. D'Souza, B.H. Kim, and R. Voyles. Morphing Bus: A rapid deployment computing architecture for high performance, resource-constrained robots. In *Robotics and Automation 2007*, pages 311–316. IEEE, 2007. doi:10.1109/ROBOT.2007.363805. Cited on pages 36 and XI.

[19] W. P. Eaton and J. H. Smith. Micromachined pressure sensors: review and recent developments. *Smart Materials and Structures*, 6(5):530, 1997. doi:10.1088/0964-1726/6/5/004. Cited on page 11.

[20] David Fischinger, Markus Vincze, and Yun Jiang. Learning grasps for unknown objects in cluttered scenes. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 609–616, 2013. doi:10.1109/ICRA.2013.6630636. Cited on page 1.

[21] FlexRay Consortium. *FlexRay Communications System. Electrical Physical Layer Specification. Version 3.0.1*, October 2010. Cited on page 25.

[22] FlexRay Consortium. *FlexRay Communications System. Protocol Specification. Version 3.0.1*, October 2010. Cited on pages 25, 41, and XI.

[23] P. Foster, A. Kouznetsov, N. Vlasenko, and C. Walker. Sub-nanosecond Distributed Synchronisation via the Universal Serial Bus. In *Precision Clock Synchronization for Measurement, Control and Communication (ISPCS) 2007*, pages 44–49. IEEE, October 2007. `doi:10.1109/ISPCS.2007.4383772`. Cited on page 46.

[24] Ricardo Franco Mendoza Garcia, Kasper Stoy, David Johan Christensen, and Andreas Lyder. A Self-reconfigurable Communication Network for Modular Robots. In *Proceedings of the 1st International Conference on Robot Communication and Coordination*, RoboComm '07, pages 23:1–23:8. IEEE Press, 2007. `doi:10.4108/ICST.ROBOCOMM2007.2156`. Cited on page 32.

[25] Daniel Gomez-Ibanez, Ethan Stump, Benjamin Grocholsky, Vijay Kumar, and Camillo J. Taylor. The robotics bus: a local communications bus for robots. In D. W. Gage, editor, *Mobile Robots XVII*, volume 5609 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pages 155–163, December 2004. `doi:10.1117/12.571476`. Cited on page 35.

[26] Matthew Grant. *Quick Start for Beginners to Drive a Stepper Motor*. Freescale Semiconductor Inc., Chandler, Arizona 85224, U.S.A., 2005. Application Note. URL: `http://www.freescale.com/files/microcontrollers/doc/app_note/AN2974.pdf`. Cited on page 13.

[27] Albert L Hopkins Jr, T Basil Smith III, and Jaynarayan H Lala. FTMP—a highly reliable fault-tolerant multiprocessor for aircraft. In *Proceedings of the IEEE*, volume 66, pages 1221–1239, October 1978. `doi:10.1109/PROC.1978.11113`. Cited on page 25.

[28] Ian P. Howard. A Note on the Design of an Electro-Mechanical Maze-Runner. *Durham Research Review*, (3), 1953. URL: `http://cyberneticzoo.com/wp-content/uploads/2009/11/Maze_IPHoward.pdf`. Cited on page 5.

[29] The Institute of Electrical and Electronics Engineers, Inc., New York, NY 10016, U.S.A. *IEEE Standard for a Smart Transducer Interface for Sensors and Actuators— Common Functions, Communication Protocols, and Transducer Electronic Data Sheet (TEDS) Formats*, 2007. `doi:10.1109/IEEESTD.2007.4338161`. Cited on page XI.

[30] InvenSense Inc., Sunnyvale, CA 94089 U.S.A. *MPU-6000 and MPU-6050 Product Specification*. Datasheet. URL: `http://www.invensense.com/mems/gyro/documents/PS-MPU-6000A-00v3.4.pdf`. Cited on page 11.

[31] Kourosh Khoshelham and Sander Oude Elberink. Accuracy and resolution of kinect depth data for indoor mapping applications. *Sensors*, 12(2):1437–1454, 2012. Cited on page 12.

[32] Uwe Koppe. TD-03011E. Identifier Usage in CANopen Networks. Technical report, January 2003. Cited on page 23.

[33] Jeff Kramer, Matt Parker, Herrera C Daniel, Florian Echtler, and Nicolas Burrus. *Hacking the Kinect*. Apress, 2012. Cited on page 12.

[34] Jukka Kyynäräinen, Jaakko Saarilahti, Hannu Kattelus, Anu Kärkkäinen, Tor Meinander, Aarne Oja, Panu Pekko, Heikki Seppä, Mika Suhonen, Heikki Kuisma, Sami Ruotsalainen, and Markku Tilli. "a 3d micromechanical compass". *Sensors and Actuators A: Physical*, 142(2):561 – 568, 2008. `doi:10.1016/j.sna.2007.08.025`. Cited on page 10.

[35] Gabriel Leen and Donal Heffernan. TTCAN: a new time-triggered controller area network. *Microprocessors and Microsystems*, 26(2):77–94, 2002. `doi:10.1016/S0141-9331(01)00148-X`. Cited on page 22.

[36] Anat Levin, Rob Fergus, Frédo Durand, and William T. Freeman. Image and Depth from a Conventional Camera with a Coded Aperture. *ACM Transactions on Graphics*, 26(3), July 2007. `doi:10.1145/1276377.1276464`. Cited on page 11.

[37] LP-RESEARCH Inc., Tokyo, Japan. *LPMS-CU Reference Manual*. Datasheet. URL: `http://www.lp-research.com/wp-content/uploads/2012/05/LpmsCUUsersGuide1.1.0.pdf`. Cited on pages 10 and 11.

[38] Olaf Lüke et al. Standalone/OnDevice Lösung. URL: `http://www.tinkerunity.org/forum/index.php/topic,2127.0.html` [cited 2014-02-19]. Cited on page 37.

[39] W. M. Macek and D. T. M. Davis. Rotation rate sensing with traveling-wave ring lasers. *Applied Physics Letters*, 2(3):67–68, 1963. `doi:10.1063/1.1753778`. Cited on page 9.

[40] Sebastian O.H. Madgwick, Andrew J.L. Harrison, and Ravi Vaidyanathan. Estimation of IMU and MARG orientation using a gradient descent algorithm. In *2011 IEEE International Conference on Rehabilitation Robotics (ICORR)*, pages 1–7, 2011. `doi:10.1109/ICORR.2011.5975346`. Cited on pages 9 and 10.

[41] Syed Tahmid Mahbub. Using the high-low side driver IR2110 - explanation and plenty of example circuits, 2013. URL: `http://tahmidmc.blogspot.com/2013/01/using-high-low-side-driver-ir2110-with.html`. Cited on page 13.

[42] MediaTek Inc., Hsinchu City 30078, Taiwan. *MediaTek-3329. 66-channel GPS Engine Board Antenna Module with MTK Chipset*. Datasheet. Cited on page 8.

[43] M. Merten and H.-M. Gross. Highly Adaptable Hardware Architecture for Scientific and Industrial Mobile Robots. In *2008 IEEE Conference on Robotics, Automation and Mechatronics*, pages 1130–1135, Sept 2008. `doi:10.1109/RAMECH.2008.4681459`. Cited on pages 35, 36, and XI.

[44] Modbus Organization, Inc., Hopkinton, MA 01748, U.S.A. *MODBUS Messaging on TCP/IP Implementation Guide. V1.0b*, 2006. URL: http://www.modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf. Cited on page 18.

[45] Modbus Organization, Inc., Hopkinton, MA 01748, U.S.A. *MODBUS over Serial Line. Specification and Implementation Guide. V1.02*, 2006. URL: http://www.modbus.org/docs/Modbus_over_serial_line_V1_02.pdf. Cited on pages 18 and 19.

[46] Modbus Organization, Inc., Hopkinton, MA 01748, U.S.A. *MODBUS APPLICATION PROTOCOL SPECIFICATION. V1.1b3*, 2012. URL: http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf. Cited on pages 18, 19, and 20.

[47] *NMEA 0183. Standard for Interfacing Marine Electronic Devices 4.00*. International Marine Electronics Association, Severna Park, MD 21146, U.S.A., 2008. Cited on page 8.

[48] Anton Nordmark. Kinect 3D Mapping. Master's thesis, Linköpings University, 2012. Cited on page 12.

[49] Roman Obermaisser. *Time-Triggered Communication*. CRC Press, Inc., 2011. Cited on pages 22 and 24.

[50] Dominique Paret. *Multiplexed Networks for Embedded Systems: CAN, LIN, FlexRay, Safe-by-Wire....* Wiley, 2007. Cited on page 20.

[51] J. Parviainen, J. Kantola, and J. Collin. Differential barometry in personal navigation. In *Position Location and Navigation Symposium (PLANS), 2008 IEEE/ION*, pages 148–152, 2008. doi:10.1109/PLANS.2008.4570051. Cited on page 10.

[52] Olaf Pfeiffer, Andrew Ayre, and Christian Keydel. *Embedded networking with CAN and CANopen*. Copperhill Technologies Corporation, 2008. Cited on page 23.

[53] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009. URL: http://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf. Cited on page 3.

[54] Lj. Ristic, R. Gutteridge, B. Dunn, D. Mietus, and P. Bennett. Surface micromachined polysilicon accelerometer. In *Technical Digest of the 5th IEEE Solid-State Sensor and Actuator Workshop*, pages 118–121, 1992. doi:10.1109/SOLSEN.1992.228311. Cited on page 10.

[55] Robert Bosch GmbH., Stuttgart, 70442 Germany. *CAN Specification. Version 2.0*, 1991. URL: http://www.bosch-semiconductors.de/media/pdf_1/canliteratur/can2spec.pdf. Cited on page 20.

[56] Robert Bosch GmbH., Gerlingen, 70839 Germany. *CAN with Flexible Data-Rate. Specification Version 1.0*, 2012. URL: http://www.bosch-semiconductors.de/media/pdf_1/canliteratur/can_fd_spec.pdf. Cited on page 21.

[57] ROBOSOFT, Bidart, 64210, France. *PURE RobuLAB Manual*, October 2012. Release 3.1. Cited on page X.

[58] Thomas Ross. Machines That Think. *Scientific American*, 148:206–209, April 1933. doi:10.1038/scientificamerican0433-206. Cited on page 5.

[59] Chavdar Roumenin. Microsensors for Magnetic Fields. In *MEMS: A Practical Guide to Design, Analysis, and Applications*, pages 453–521. Springer Berlin Heidelberg, 2006. doi:10.1007/978-3-540-33655-6_9. Cited on page 10.

[60] Behnam Salemi, Mark Moll, and Wei-Min Shen. SUPERBOT: A Deployable, Multi-Functional, and Modular Self-Reconfigurable Robotic System. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3636–3641, October 2006. doi:10.1109/IROS.2006.281719. Cited on page 32.

[61] U. Saranli, A. Avci, and Andztü Andrk. A Modular Real-Time Fieldbus Architecture for Mobile Robotic Platforms. *IEEE Transactions on Instrumentation and Measurement*, 60(3):916–927, March 2011. doi:10.1109/TIM.2010.2078351. Cited on page 35.

[62] Semiconductor Components Industries, LLC, 2009, Denver, CO 80217 U.S.A. *AMIS-30660/42000 - Topology Aspects of a High-Speed CAN Bus*, 2009. Application Note. URL: https://www.onsemi.com/pub/Collateral/AND8376-D.PDF. Cited on page 19.

[63] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to Autonomous Mobile Robots*. The MIT Press, 2nd edition, February 2011. Cited on page 7.

[64] Telecommunications Industry Association, Arlington, VA 22201, U.S.A. *Application Guidelines for TIA/EIA-485-A*, 2006. Cited on page 17.

[65] u-blox A.G., 8800 Thalwil, Switzerland. *UP501. Fastrax GPS patch antenna module*. Datasheet. URL: http://www.u-blox.com/images/downloads/Product_Docs/UP501_DataSheet_(FTX-HW-12010).pdf. Cited on page 8.

[66] USB Implementers Forum, Inc., Beaverton, OR 97006, U.S.A. *Universal Serial Bus Specification. Revision 2.0*, 2000. URL: http://www.usb.org/developers/docs/usb20_docs/usb_20_012314.zip. Cited on pages 26, 29, and 45.

[67] V. Vali and R. W. Shorthill. Fiber ring interferometer. *Applied Optics*, 15(5):1099–1100, May 1976. doi:10.1364/AO.15.001099. Cited on page 9.

[68] VectorNav Technologies LLC., Dallas, TX 75238 U.S.A. *VN-200 User Manual*. Datasheet. URL: http://www.vectornav.com/Downloads/Support/UM004.pdf. Cited on pages 9 and 11.

[69] Richard A. Wallace. The maze solving computer. In *Proceedings of the 1952 ACM National Meeting (Pittsburgh)*, pages 119–125. ACM, 1952. doi:10.1145/609784.609800. Cited on page 5.

[70] M. Wargui and A. Rachid. Application of Controller Area Network to Mobile Robots. In *8th Mediterranean Electrotechnical Conference*, volume 1, pages 205–207, 1996. doi:10.1109/MELCON.1996.550992. Cited on page 34.

[71] Xsens Technologies B.V., Enschede, 7500 AN The Netherlands. *MTi User Manual (MT0605P)*. Datasheet. URL: http://www.xsens.com/download/usermanual/MTi_usermanual.pdf. Cited on pages 8 and 11.

[72] Jorge Zambada. *Measuring Speed and Position with the QEI Module*. Microchip Technology Inc., Chandler, AZ 85224 U.S.A., 2005. Application Note. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/93002A.pdf. Cited on page 8.

[73] Jieying Zhang, E. Edwan, Junchuan Zhou, Wennan Chai, and O. Loffeld. Performance investigation of barometer aided GPS/MEMS-IMU integration. In *Position Location and Navigation Symposium (PLANS), 2012 IEEE/ION*, pages 598–604, 2012. doi:10.1109/PLANS.2012.6236933. Cited on pages 9 and 10.

[74] Kai Zhou, Karthik Mahesh Varadarajan, Michael Zillich, and Markus Vincze. Spatial structure analysis for autonomous robotic vision systems. In *IEEE Workshop on Robot Vision (WORV)*, pages 165–170, 2013. doi:10.1109/WORV.2013.6521933. Cited on page 1.

Acronyms

8P8C 8 positions 8 contacts, often referred to as RJ45 (pp. 35, 42)
A2J arduino2java (pp. 51, 54, 55, 58, 68)
ABI application binary interface (p. 52)
ACIN Automation and Control Institute (pp. 1, 3)
ADC analog-to-digital converter (pp. 4, 40)
AFDX Avionics Full-Duplex Switched Ethernet (p. 17)
AHRS attitude and heading reference system (pp. 1, 4, 9–11, 16, 41)
API application programming interface (pp. 38, 48, 51, 55)
ARINC Aeronautical Radio, Incorporated (p. 17)
ARM Advanced RISC Machine (pp. 37, 38)
ASCII American Standard Code for Information Interchange (pp. 8, 18)
BRS Bit Rate Switch (p. 21)
BT Bluetooth (pp. 8, 35, 44, 51)
CAN Controller Area Network (pp. VIII, X, 4, 11, 17, 19–23, 34–36, 39–43)
CAN FD CAN with Flexible Data-Rate (p. 21)
CAS Collision Avoidance Symbol (p. 24)
CC communication controller (pp. 24, 25)
CDC Communication Device Class (p. 51)
CERN Conseil Européen pour la Recherche Nucléaire (French for "European Council for Nuclear Research") (p. 38)
CiA CAN in Automation (pp. IX, 22, 23, 42)
CPR counts per revolution (p. 7)
CPU Central Processing Unit (p. 36)
CRC cyclic redundancy check (pp. 18, 21, 24, 25, 27, 36, 46)
CSMA/CR Carrier Sense Multiple Access with Collision Resolution (p. 20)
DOF degrees of freedom
DS Draft Standard (p. 23)
DSP Draft Standard Proposal (p. 23)
DTS Dynamic Trailing Sequence (p. 24)
ECN engineering change notice (p. 26)
EDS electronic data sheet (pp. 22, 33)
EN European Standard (from German "Europäische Norm") (p. 22)
EOF End-of-Frame (p. 27)
EOL end-of-life (p. 17)
EOP End-of-Packet (p. 27)
FIFO first-in, first-out (p. 20)
FOG fiber optic gyroscope (p. 9)
FPGA field-programmable gate array (pp. 36, 37, 43)
GNSS global navigation satellite system (pp. 8, 9)
GPIO general-purpose input/output (pp. 4, 40)
GPL GNU General Public License (p. 38)
GPS Global Positioning System (pp. 1, 4, 8, 9, 11, 16, 38, 41)
HDOP Horizontal Dilution of Precision
HomePNA Home Phone Networking Alliance (p. 34)
I^{2}C Inter-Integrated Circuit (pp. 4, 6, 7, 11, 35, 36, 38, 40, 43)
IEC International Electrotechnical Commission (pp. 17, 42)
IEEE Institute of Electrical and Electronics

Engineers (pp. 33, 34)
IMU inertial measuring unit (pp. 9, 11, 16, 35, 41)
INS inertial navigation system (p. 9)
IP Internet Protocol (pp. 18, 38)
IR infrared (pp. 3–7, 11, 12, 16, 32, 41)
ISO International Organization for Standardization (pp. 21, 23)
ISR interrupt service routine (pp. 48, 50, 54, 68, 71)
ITU International Telecommunication Union (p. 34)
J2A java2arduino (pp. 51–54, 56–58, 60, 62–66, 71, 73, 74)
L2CAP Logical Link Control and Adaptation Protocol (p. 51)
LCD liquid-crystal display (p. 38)
LED light-emitting diode (p. 32)
LIDAR light radar (pp. 4, 11, 12, 16, 41)
LIN Local Interconnect Network (pp. 17, 43)
LRC longitudinal redundancy check (p. 18)
LSB least significant bit (p. 29)
LSS Layer Setting Services, CiA 305 (p. 23)
LUFA Lightweight USB Framework for AVRs (p. 71)
MARG magnetic, angular rate, and gravity (pp. 9, 11)
MEI MODBUS Encapsulated Interface (pp. 19, 20)
MEMS microelectromechanical system (pp. 9–11)
MSB most significant bit (p. 29)
NCAP Network-Capable Application Processor (pp. 33, 34)
NIR near-infrared (p. 12)
NIT network idle time (pp. 23, 24)
NMEA National Marine Electronics Association (pp. 8, 9, 11, 21)
NMO Network Management Object (p. 22)
NMT Network Management (p. 23)
NRZI non-return to zero inverted (p. 27)
OD Object Dictionary (pp. 22, 23, 35, 43)
OSI Open Systems Interconnection (p. 33)
PC Personal Computer (p. 26)
PCB printed circuit board (pp. 36, 38)
PD public domain
PDO Process Data Object (pp. 22, 23, 35)
PDU Protocol Data Unit (p. 18)
PID Packet ID (pp. 27–29)
PoE Power over Ethernet (pp. 26, 38)
POSIX Portable Operating System Interface (p. 48)
PPP Point-to-Point Protocol (p. 52)
PSD Position Sensitive Device (pp. 5, 6)
PWM pulse width modulation (pp. 13, 14, 39, 40, 72)
RAII Resource Acquisition Is Initialization (p. 66)
RED Rapid Embedded Development (p. 37)
RFC Request For Comments (p. 52)
RFCOMM Radio Frequency Communication (p. 51)
RFID radio-frequency identification (p. 34)
RGB-D RGB-Depth/Distance (pp. 4, 12, 16, 40, 41)
RISC Reduced Instruction Set Architecture (p. VIII)
RLG ring laser gyroscope (p. 9)
ROS Robot Operating System (pp. 3, 39, 43, 44, 60, 63–65, 68, 69, 71, 73)
RPC remote procedure call (p. 44)
RS Recommended Standard (p. 17)
RTC real-time clock (pp. 44, 50)
RTU Remote Terminal Unit (pp. 18, 38)
SAE Society of Automotive Engineers (pp. 17, 21)
SBC single-board computer (p. 39)
SCADA supervisory control and data acquisition (p. 33)
SDO Service Data Object (pp. 22, 23)
SE0 Single-ended 0 (p. 27)
SE1 Single-ended 1 (p. 27)
SFO Special Function Object (p. 22)
SIF server-initiated frame (pp. 53, 54, 58, 61, 65, 66)
SLAM Simultaneous Localization and Map-

ping (p. 1)
SOC system on chip (pp. 39, 40)
SOF Start-of-Frame (pp. 27, 44, 45, 48, 50, 74)
SPI Serial Peripheral Interface (pp. 4, 11, 34, 38, 40)
SPP Serial Port Profile (p. 51)
SPS Standard Positioning Service (p. 9)
STIM Smart Transducer Interface Module (pp. 33, 34)
TCP Transmission Control Protocol (pp. 18, 38)
TDMA time-division multiple access (p. 23)
TEDS Transducer Electronic Data Sheet (pp. 33, 34)
TIA Telecommunications Industry Association (pp. 4, 17–19, 35, 38, 40, 43, 51)
ToF Time-of-Flight (pp. 4, 11, 12, 16, 40, 41)
TTCAN Time-Triggered CAN (pp. 21, 22)
TTL Transistor-Transistor Logic (pp. 4, 7, 8, 40)
TTP Time-Triggered Protocol (pp. 17, 22)
URB Universal Robot Bus (pp. 35, 36)
US ultrasonic (pp. 4, 6, 16, 41)
USB Universal Serial Bus (pp. IX, XI, 7, 8, 11, 26–29, 31, 34, 35, 38, 40–46, 48, 50, 51, 58, 71, 73, 74)
UTC Coordinated Universal Time (p. 9)
WG working group (p. 34)
WGS World Geodetic System (p. 9)
WLAN Wireless Local Area Network (pp. 35, 38)

List of Figures

1.1 Abstract schematic of a typical off-the-self robot (ROBOSOFT robuLAB10) taken from its manual [57]. 2

2.1 Working principle of optical distance sensors (own work). 6
2.2 Picture of a Sharp GP2Y0A02YK IR distance sensor (own work). 6
2.3 Working principle of sonar ranging by Georg Wiora (GFDL 1.2+ and CC-BY-SA-2.5+), via Wikimedia Commons. 7
2.4 Example output of a quadrature encoder by Sagsaw [CC0] (public domain (PD)), via Wikimedia Commons. 8
2.5 Photo of the two dies contained in an Invensense MPU6050 published by ZeptoBars (CC-BY 3.0), via http://zeptobars.ru. 10
2.6 Schematic of an H bridge slightly adapted from Cyril Buttay (PD), via Wikimedia Commons. 13
2.7 Schematic cross section of a stepper motor by “Stündle” (PD), via Wikimedia Commons. 14
2.8 PWM control of a servo motor (own work). 15

3.1 RTU message format (own work). . . . 18
3.2 ASCII message format (own work). . . . 18
3.3 Frame formats in CAN (own work). . . . 21
3.4 Structure of FlexRay communication adapted from Ralf Pfeier (GDL 1.2+, CC-BY-SA 3.0), via Wikimedia Commons. . . . 24
3.5 FlexRay frame format adapted from the FlexRay specification [22]. . . . 25
3.6 FlexRay topology examples adapted from the FlexRay specification [22]. . . . 26
3.7 USB framing (own work). . . . 28
3.8 Main types of USB packets (own work). . . . 28
3.9 USB device architecture (own work). . . . 30

4.1 A *simple* overview of IEEE 1451 (own work) inspired by Figure 2 of [29]. . . . 33
4.2 Schematic of one example using the adaptable architecture presented in [43] taken from the paper. . . . 36
4.3 Diagram of the connection principle of the Morphing Bus (adapted from [18]). . . . 37
4.4 A stack of Tinkerforge Bricks taken from the Tinkerforge website (CC-BY-SA 3.0). 38

6.1 Synchronization scheme (own work), clock by Jahoe (CC-BY-SA 3.0, original design probably copyrighted too) and USB logo (by Chad Weider, probably PD) via Wikimedia Commons. . . . 46

7.1 Format of J2A packets (without Byte Stuffing) (own work). . . . 52
7.2 Microcontroller side of J2A (dubbed A2J) (own work), Bluetooth logo (by Jnmasek, probably PD) and USB logo (by Chad Weider, probably PD) via Wikimedia Commons. . . . 55
7.3 Host side of J2A (own work), Bluetooth logo (by Jnmasek, probably PD) and USB logo (by Chad Weider, probably PD) via Wikimedia Commons. . . . 59

8.1 Overview of the prototype system (own work), USB logo (by Chad Weider, probably PD) via Wikimedia Commons. . . . 61
8.2 Overview of the firmware architecture based on the generated code (own work), USB logo (by Chad Weider, probably PD) via Wikimedia Commons. . . . 63
8.3 Overview of the proxy interface for ROS (i.e. *missing_link*) (own work), USB logo (by Chad Weider, probably PD) via Wikimedia Commons. . . . 65

9.1 Test setup (own work). . . . 72