Design of an Open Hardware Bridge Between Robots and ROS / non–ROS Environments

Yves BERGEON[1], *Vaclav KRIVANEK*[2]

[1]Center of Research “Ecoles de Saint–Cyr Coetquidan“, 56381 GUER Cedex, France
[2]Department of Air Defence Systems, Faculty of Military Technology, University of Defence, Kounicova 65, 612 00 Brno, Czech Republic

yves.bergeon@st-cyr.terre-net.defense.gouv.fr, vaclav.krivanek@unob.cz

DOI: 10.15598/aeee.v16i2.2439

Abstract. *To be able to use robots of the same type in a swarm (in this case differential–drive robots) from a centralized computer, you need to control them in the same way even if the robots come from different manufacturers. These robots should have the same capabilities: i.e. the same type of sensors and the same type of control commands. Currently, if we need to add sensors on a robot, we should first find out what are the hardware capabilities and what are the tools provided by the manufacturer (some have microcontrollers, others have ARM processors). The protocol to communicate with the onboard card (often a serial protocol) is different for each robot and we have to take that in account in our software so that the robots can be interchangeable.*

This paper describes the design and the results of using a standard interface (bridge) on a robot which allows wireless communication with a centralized computer and local serial communication with a mobile robot. On this interface, we can connect every sensor we want (infrared, ultrasound, laser, Kinect etc.) and we can always use the same tools to control them.

Extension of the protocol of the Kobuki base (TurtleBot 2) is proposed to be able to use the same type of frame as the TurtleBot 2 uses and to continue to use ROS middleware as is, but with the possibility to use data from new sensors implemented on the bridge. This article will focus on the use of the TurtleBot 2, but the same bridge will be adapted to other robots allowing their interchangeability in the future.

Keywords

Differential–drive robot, interface, robotics, ROS, swarm, Turtlebot 2, unicycle robot.

1. Introduction

Many works, allowing interfacing between different robot environments, have been done in the last decade. Microsoft Robotics Developer Studio (deprecated now) and Robot Operating System (ROS) were introduced. ROS is now widely used but some works continue to exist without ROS. The biggest advantage of ROS is the large independence of the work for each part (drivers, middleware [1], etc.) using asynchronous transmission (topics for ROS).

For hardware environment, the initiative of the Hardware Robot Operating System (H–ROS) is a very interesting option for building interoperable robot components [2]. But this initiative is very young and the components are not yet on the market. Currently, interfaces for sensors and actuators are very different: analog voltage, I^2C, SPI etc.

When we need to control robots from different manufacturers in a same way (i.e. in a swarm) we should deal with the problem that each robot has its own on–board sensors. If we want to add new sensors on a robot, we need to learn how the on–board controller card works, what tools there are to program it (some needs to use cross–compiler on a specific computer) and to see if adding new sensors is possible or not.

This paper describes the design of an interface which is used as a bridge between the on–board controller on each of our robots and a centralized computer elsewhere. That is even though each of our robots is from a different manufacturer with different interfacing specifics. We chose to use open solutions with the large support of the community for the hardware and to propose an extension of the protocol of communication of the Kobuki base (TurtleBot 2) allowing us to use the same protocol for all of our robots.

Kobuki base of TurtleBot 2 robot is a widely used open source hardware project [3]. TurtleBot 2 has found use in many domains, ranging from research of robotics [4] through research in social sciences, to application in service robotics [5]. This robot also plays an important role in teaching at universities and high schools [6]. Industrial applications can also be found. Kobuki is supported by many environments like ROS, Gazebo simulator [7], V–REP simulator and others. Even Matlab supports Kobuki thanks to Robotics toolbox.

This paper is divided into five sections. Section 2. contains a description of problems related to the multi–robot swarm control. Section 3. shows possible solutions to the interface (a bridge) using different hardware technologies. Section 4. deals with the hardware part we defined and with the extension of TurtleBot 2 communication protocol. And finally, Sec. 5. contains summarized conclusion of the paper and a plan for a future work on this topic.

2. Problem Definition

We need to control different kinds of the differential–drive robot: *Eddie Robot* from Parallax, Inc., *TurtleBot 2* with Kobuki base and *Khepera IV* from EPFL company, and to use all of them as a swarm, in ROS or non–ROS environment (e.g. MATLAB). The purpose of the project is to solve problems (e.g. environment mapping, swarm formation control [8]) with real robots and in real conditions.

Even if the characteristics of these robots are different (size, weight), they have comparable maximum speed. Sensors onboard are different, but we want to be able to add the same kinds of sensors on each robot.

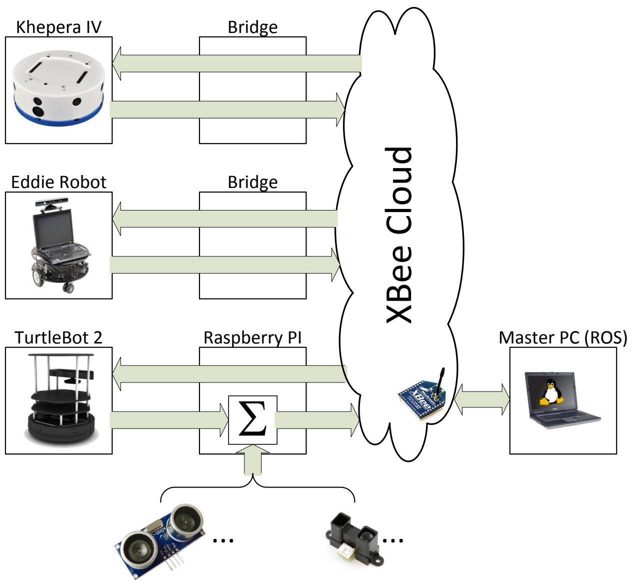


Fig. 1: Swarm organization, robot's communication via XBee.

The heart of our system is a centralized computer which controls moves of all of the robots in a swarm (see Fig. 1). The communication needs to be wireless. To control low–level tasks (communication with sensors or actuators) on our robots (Eddie and TurtleBot 2) we currently use computers under Linux and a printed circuit with Microchip PIC microcontroller. But on small robots like Khepera IV, it is impossible to add the heavy weight of the computer. Moreover, the cost is very high because the computers on our robots are used only for communication between robots, microcontrollers and the centralized computer.

3. Solution Analysis

3.1. Constraints

We want to add a low–cost system which allows communication with all sensors in the same way on each of our robots. This system needs to be widely used for the drivers to be obtainable from the community and it also needs to have enough computing power to be able to execute all the needed tasks at a low energy cost. Moreover, this system needs to have implemented emergency situation behaviours (failure of communication with centralized computer).

Wireless communication is also needed. Our robots move in different kinds of environment where Wi–Fi is not always available. The choice was made to use Wi–Fi only to transmit images or videos if available.

The main communication protocol remains a serial communication, implemented with wireless technology. Different wireless technologies have been tested before implementing on our robots. Lora (Long range) network was evaluated but latency is too high and bandwidth too small for the needs of robots' communication and control. Nordic nRF24L01+ had also be evaluated and could be an interesting solution for replacing XBEE, but is less widely available than XBEE.

In this article, authors will only focus on the use of Digi XBEE for communication, which allows a point to point (or multiple points depending on the version) serial half–duplex communication at 115 200 bauds between our centralized computer and our robots.

3.2. Interface (Bridge)

The card we want to add needs to be able to communicate transparently with the robot and the centralized computer: we need *a bridge*. The first goal of the bridge is to communicate locally with the robot and to send data to the centralized computer and to do the same in the other direction.

But we have a second goal: the bridge must be able to communicate with any new sensors that were applied. We want, of course, the values of the sensors to be sent in the same way no matter if they are interfaced on the robot itself or if they are interfaced on the new card itself.

Several projects have been done on this topic, for example, *rosserial* [9]. This work was initially done on Arduino card to communicate through XBEE transmitter / receiver in part of the ROS environment. Another work *rosbridge* [1] allows communication between ROS and non–ROS environment. In case of *rosserial*, the implementation is done on Arduino which can not get our minimal requirement for computing power. And none of the other works is an answer to our problem to work both in ROS and non–ROS environments.

For every robot we bought, the manufacturer has implemented controls for all of the sensors and actuators through a serial protocol included in a USB communication (115 200 bauds, full duplex). Our goal is to be able to allow these communications from/to the centralized computer through an XBEE (serial protocol). But our system will also interface with new sensors, reading values from them and multiplex these values in the serial communication and send them to the centralized computer.

3.3. Bridge Choice

Authors' idea is to use an open–source hardware card, with enough computing power (with an option of future versions, if possible), widely available and with a strong community. For our needs, the Raspberry Pi 3 was chosen to be used. The community is large, the cost is low and there are many interfaces including a standardized interface: the Global Purpose Input / Output (GPIO).

We also have USB interfaces (which is very important not only for our serial communications but also for some laser sensors) and Ethernet interface (which can be used for some laser sensors as well). For GPIO, we have direct control of digital inputs and outputs. And the serial, SPI and I^{2}C protocols are implemented on it as well. The last two mentioned protocols are often used to interface with sensors.

The most notable downside of this card is the absence of an Analog to Digital Converters (ADC) needed for some infrared or laser sensors. The desired update of this card will arrive with the next version of the Raspberry Pi (more computing power and memory), but it should be noted that a card of the same format with the same GPIO is offered by many competitors (e.g. Asus Tinker board).

4. Hardware Design and Communication Protocol

4.1. The Robot's Hat

One big advantage of the Raspberry Pi is its standard GPIO that is used by many COTS (Commercial off–the–Shelf) projects, e.g. the *Sense Hat* [10].

But none of these projects fulfills our needs completely. For that reason, we instead developed a new card, which we named "Robot's hat". This card will come on top of the Raspberry Pi using the GPIO connector (see Fig. 2). Our goal is for the card to have only one type of printed circuit while it would be usable by all of our robots. Ultrasound and infrared sensors are included on Eddie and Khepera IV, but not on TurtleBot 2. The Inertial Measurement Unit (IMU) is then on Khepera IV and TurtleBot 2, but not on Eddie. And laser sensors aren't included on any of these robots.

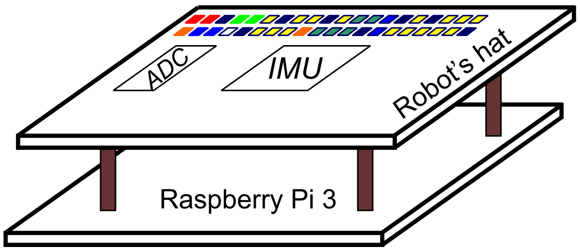


Fig. 2: The robot's hat on top of the Raspberry Pi.

The printed circuit must have:

- I^{2}C interface for sensors like ultrasound Devantech SRF08; an I^{2}C connector with eight entries on the board (it allows direct connection through three pins — serial clock, serial data and ground).
- ADC converter to convert analog voltage from sensors like infrared Sharp GP2Y0A21YK; we chose the Microchip MCP3008 allowing us to convert up to eight analog inputs and to communicate with Raspberry Pi by the I^{2}C protocol (avoiding using other pins on the GPIO connector).
- Direct digital interface for sensors like Ping from Parallax; a connector with eight entries is available on the card (three pins on top are used if sensor works in 3.3 V and the three pins on the bottom are used if sensor works in 5 V).
- Designated space in its centre for the possible need of an IMU implementation. Microchip MM7150 was selected as the preferred IMU and it includes a three–axis accelerometer, gyroscope and magnetometer pre–programmed with integrated calibration and sensor fusion algorithms. The protocol used by this IMU is also I^{2}C, and so can be interfaced directly to the Raspberry Pi.

Tab. 1: Kobuki base bytestream.

Name	**Header 0**	**Header 1**	**Length**	**Payload**	**Checksum**
Size [Byte]	1	1	1	N	1
Description	0xAA (Fixed)	0x55 (Fixed)	Size of payload in bytes	Described below	XOR'ed values of every byte of the bytestream, except headers

4.2. Analysis of the Communication Protocol of TurtleBot 2

The communication on all of our robots is done through serial protocol over USB. The bridge will communicate to the robot locally and the communication between the robot and the centralized computer should be as transparent as possible, see the bottom part of Fig. 1.

The communication works differently on each of the considered robots. On Khepera IV and Eddie, to get the sensor values appropriate commands have to be sent from the computer. On the other hand, the TurtleBot 2 sends the value of their on–board sensors (bumpers, cliff sensor, IMU etc.) automatically 50 times per second in a bytestream (frame), see Tab. 1.

This frame is divided into different parts:

- Two first bytes of each frame are in hexadecimal: 0xAA 0x55. Our software must find these two values to be sure to get the beginning of a frame.
- The 3rd byte defines the length of the *payload* (size of the rest of the frame except checksum). As the 3rd byte contains 8 bits, we can have 255 bytes for the payload, so 259 bytes at maximum for a frame (255 bytes of payload + 2 bytes for header + 1 byte for length + 1 byte for checksum).
- The payload is divided into sub–payloads (see Tab. 2). List of the used payloads can be found in Tab. 3. As we can see, some of the sub–payload IDs are reserved, allowing us to create new sub–payloads later down the line.
- The last byte is a checksum to verify the integrity of the data.

We want to be able to use ROS as well. TurtleBot 2 is widely used and interfaces well with ROS Middleware on computers. Our idea was to use the same protocol as the one implemented on TurtleBot 2 to transmit values of the different sensors on all of our robots, except for Kinect that will use Wi–Fi if available (the bandwidth needed is too high for image transmission).

Tab. 2: Structure of sub–payloads.

Name	**Header**	**Length**	**Data**
Size [Byte]	1	1	N
Description	Predefined identifier	Size of data in byte(s)	Described below

Tab. 3: Structure of feedback packets.

ID	**Description**
1	Basic core sensor data
2	Reserved
3	Signals from docking station
4	Gyro data both angle and angular velocity
5	PSD data facing floor
6	Current of wheel motors
7	Reserved
8	Reserved
9	Reserved
10	Version number of Kobuki hardware
11	Version number of Kobuki hardware
12	Reserved
13	Raw ADC data of digital 3–axis gyro
14	Reserved
15	Reserved
16	Inputs from 25–pin expansion port
17	Reserved
18	Reserved
19	Unique number to identify robot
20	Reserved
21	PID gain value of wheel velocity controller

4.3. Protocol Design

1) Frame Size and Frequency

Turtlebot 2 sends 50 frames per second by default. If we send the maximum size of the frame at 50 Hz, we will get 129500 $\text{bit}\cdot\text{s}^{-1}$ which exceeds the one way capacity of our serial protocol(half duplex wireless communication) by 12 %. We also need to preserve bandwidth in the other way to be able to send commands from centralized computer to the robot.

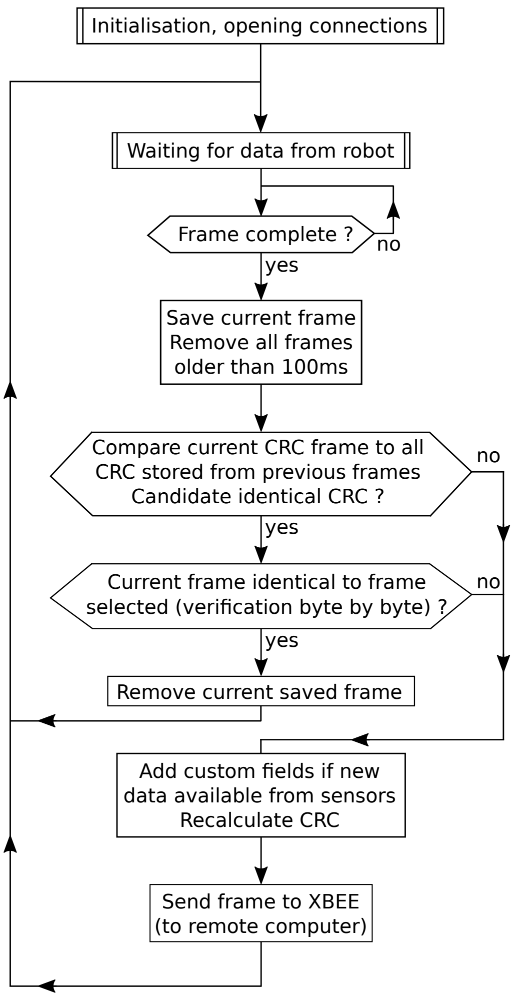
2) Protocol Implementation

We chose to use the Python language because most of the sensor drivers from the community are in Python. For communication, we use the serial module for Python which allows us to get a bi–directional serial communication both to the robot and to the XBEE module.

To be sure that no bytes are lost on arrival, we used threads in our Python implementation. The implementation works in this way:

- Two threads are used to verify if bytes arrive from the XBEE (i.e. from the centralized computer) or if the bytes arrive from the robot on the serial link.

- If data arrive from the computer, the data are stored in a buffer, the checksum is verified, and in the case the checksum matches, the frame is directly sent to the robot to avoid latency.
- If data arrive from the robot, they are stored in a buffer and checksum is verified. This frame is then compared with all of the previous frames arrived in the last 100 ms and if it's a new frame, we send the frame with new fields (see below) to the computer. Each frame sent to the computer contains a sub–payload with the robot's unique number, allowing the computer to find out from which robot the data received comes from (see Tab. 4 and Tab. 2). If an exact frame was sent in the last 100 ms, we throw the new one away (to lower the frequency down to 10 Hz). See Fig. 3 for the algorithm.
- One thread is implemented to read the values of the sensors connected directly on the Raspberry Pi. When new values are available, they are added to the frame sent to the computer using one of the reserved IDs that hasn't been used yet.



Initialisation, opening connections
Waiting for data from robot
Frame complete ?
no
yes
Save current frame
Remove all frames
older than 100ms
Compare current CRC frame to all
CRC stored from previous frames
Candidate identical CRC ?
no
yes
Current frame identical to frame
selected (verification byte by byte) ?
no
yes
Remove current saved frame
Add custom fields if new
data available from sensors
Recalculate CRC
Send frame to XBEE
(to remote computer)

Fig. 3: Data manipulation algorithm.

To be able to add new data to the frame with no effect on ROS middleware, we must define new IDs that will be ignored by the nodes currently used. The new IDs that we have defined are some of the reserved IDs in the Kobuki protocol (in hexadecimal), see Tab. 4.

The sub–payloads are then defined by using another similar sub–payload of the Kobuki protocol, see Tab. 5.

Tab. 4: Structure of feedback packets.

ID	Data description
0x02	Number of the robot
0x81	Ultrasound data
0x82	Infrared data
0x83	Laser data

Tab. 5: Structure of feedback packets.

Description	Size
Feedback identifier	1 byte
Nb of frames needed for a complete transmission (only needed for laser 0x83, because data can exceed frame size)	1 byte
Size of the data field of this sub–payload	1 byte
Size of each value	1 byte (value = 1 for uint8, = 2 for uint16, and 4 for unit32)
Timestamp	2 bytes
Data of the sensors	1 to 192 bytes

The implementation of the protocol in Python is finished but works only partially. We have developed it in Python language, using threads and queues to communicate between the threads. If the transmission is implemented only in one way (data values from sensors + frame from Kobuki) using two threads, then the system works great. But if threads for the other way are added, then the performance significantly decreases (very high CPU usage and loss of some frames). This problem is due of the implementation of GIL (Global Interpreter Lock) in python which limits the use of only one core on the CPU (instead of four on the Raspberry Pi 3). A partial reimplementation of our system is currently done using the multiprocessing module (tasks will be used instead of threads).

5. Conclusions

In this paper, we present our design of a standard interface for robots from different manufacturers. Its goal is for our centralized computer to communicate in the same way with all of our robots (all of them are differential–drive robots from different manufacturers).

The core of our system is a widely used computer card: the Raspberry Pi version 3. On top of this, we have added connectors for different kinds of sensors (infrared, ultrasound, laser) in different protocols (I^2C, SPI, digital and analog inputs) allowing us to add all necessary sensors if not included on the robot. The interface to communicate between sensors and the ARM processor on the Raspberry Pi uses Python language.

Our work is not focused only on the hardware part but also on the communication protocol. To be able to reuse middleware ROS, we have added some extensions to the Kobuki base protocol (TurtleBot 2) allowing us to communicate with ROS nodes as usual. In a future work, we will add ROS nodes to be able to use our frames which include values of the added sensors.

The next step is to use the Raspberry Pi on our other robots (Khepera IV and Eddie) and to program it to get values from sensors at 10 Hz (contrary to the TurtleBot 2 which sends sensor values on its own). The Raspberry Pi also needs to be programmed to translate orders of the centralized computer (which are the same commands as for the TurtleBot 2) to commands understandable by all of these different kinds of robots. The following step is then to automatically detect to which robot is the Raspberry Pi connected and appropriately change the program. The goal is to have a swarm of robots from different manufacturers and be able to use each of them in the same way from a centralized computer. To allow this, we have added in the frame a unique number for identification of each robot, which is a great improvement for using robots in swarm, especially in ROS environment.

Acknowledgment

The work presented in this article has been supported by CREC St–Cyr Coetquidan (France) and the Czech Republic Ministry of Defence (University of Defence development program "Research of sensor and control systems to achieve battlefield information superiority“).

References

[1] CRICK, C., G. JAY, S. OSENTOSKI, B. PITZER and O. C. JENKINS. Rosbridge: ROS for Non-ROS Users. In: *Robotics Research: The 15th International Symposium*. Flagstaff: IFRR, 2011, pp. 493–504. ISBN 978-3-319-29362-2. DOI: 10.1007/978-3-319-29363-9_28.

[2] The Hardware Operating System. In: *H-ROS* [online]. 2018. Available at: `https://www.h-ros.com/`.

[3] TurtleBot2. In: *Turtlebot.com* [online]. 2017. Available at: `https://www.turtlebot.com/turtlebot2/`.

[4] BERGEON, Y. T., I. HADDA, V. KRIVANEK, J. MOTSCH and A. STEFEK. Low cost 3d mapping for indoor navigation. In: *International Conference on Military Technologies*. Brno: IEEE, 2015, pp. 689–693. ISBN 978-8-0723-1977-0. DOI: 10.1109/MILTECHS.2015.7153749.

[5] GEORGOULAS, C., T. LINNER, A. KASATKIN and T. BOCK. An AmI Environment Implementation: Embedding TurtleBot into a novel Robotic Service Wall. In: *7th German Conference on Robotics*. Munich: VDE, 2012, pp. 1–6. ISBN 978-3-8007-3418-4.

[6] RUZZENENTE, M., M. KOO, K. NIELSEN, L. GRESPAN and P. FIORINI. A Review of Robotics Kits for Tertiary Education. In: *3rd International Workshop Teaching Robotics, Teaching with Robotics*. Riva del Garda: Citeseer, 2012, pp. 153–162. ISBN 978-88-95872-05-6.

[7] KUMAR, N., Z. VAMOSSY and Z. M. SZABO-RESCH. Robot path pursuit using probabilistic roadmap. In: *17th International Symposium on Computational Intelligence and Informatics*. Budapest: IEEE, 2016 pp. 139–144. ISBN 978-1-5090-3909-8. DOI: 10.1109/CINTI.2016.7846393.

[8] GALLARDO, N., K. PAI, B. A. EROL, P. BENAVIDEZ and M. JAMSHIDI. Formation control implementation using Kobuki TurtleBots and Parrot Bebop drone. In: *World Automation Congress*. Rio Grande: IEEE, 2016, pp. 1–6. ISBN 978-1-8893-3551-3. DOI: 10.1109/WAC.2016.7582996.

[9] Rosserial. In: *ROS wiki* [online]. 2018. Available at: `http://wiki.ros.org/rosserial`.

[10] Sense Hat. In: *Raspberry Pi* [online]. 2018. Available at: `https://www.raspberrypi.org/products/sense-hat`.

About Authors

Yves BERGEON was born in France. He received his Ph.D. from University of Paris Sud Orsay, Orsay, France in 1996. His research interests include robotics and control of systems.

Vaclav KRIVANEK was born in Brno, Czech Republic. He received his M.Sc. in specialization The Control and Guidance Systems of Missiles from Military Academy, Brno, Czech Republic in 2002. In 2006 he obtained Mastere specialise of Techniques for Aeronautics and Space from SUPAERO, Toulouse, France. In 2010 he received the Ph.D. degree in diagnostic methods from University of Defence, Brno, Czech Republic. His research interests include feedback control systems applied to air defence.