High-Frequency Multi Bus Servo and Sensor Communication Using the Dynamixel Protocol

Marc Bestmann, Jasper Güldenstein, and Jianwei Zhang

Hamburg Bit-Bots, Department of Informatics, University of Hamburg,
Vogt-Kölln-Straße 30, 22527 Hamburg, Germany
{bestmann, 5guelden, zhang}@informatik.uni-hamburg.de
http://robocup.informatik.uni-hamburg.de

Abstract. High-frequency control loops are necessary to improve agility and reactiveness of robots. One of the common limiting bottlenecks is the communication with the hardware, i.e., reading of sensors values and writing of actuator commands. In this paper, we investigate the performance of devices using the widespread Robotis Dynamixel protocol via an RS-485 bus. Due to the limitations of current approaches, we present a new multi-bus solution which enables typical humanoid robots used in RoboCup to have a control loop frequency of more than 1 kHz. Additionally, we present solutions to integrate sensors into this bus with high update rates.

Keywords: robotics, humanoid, servo, sensor, control, bus, open source

1 Introduction

Daisy chained servo motors with bus communication are an essential component of many robots. They reduce the complexity of the mechanical system by minimizing the number of cables required for controlling the actuators. Especially in robots with many degrees of freedom, i.e., humanoids, it is impractical to control motors in parallel. Daisy chaining sensors into the bus can further reduce the number of needed cables.

A widely used standard for peripheral communication is RS-485 (also known as TIA-485 or EIA-485) [9]. Robust communication in electrically noisy environments is achieved with a differential signal. One widely used servo motor series using RS-485 for communication are Dynamixel servos by Robotis[1]. They are used in a wide range of scenarios from robot arms and end effectors to small to full-size humanoids. Some of these applications are described in Section 3.

Fast and reliable communication with the actuators is essential for most use cases. Since lower latency is synonymous with a faster response to sensory

[1] Some subseries (e.g., the MX or X series) are also available with a buffered TTL interface as a physical communication layer. Servos with a TTL interface can be connected to the same hardware as ones with an RS-485 interface as explained in Section 2.

inputs, motions can be more dynamic and reactive. This is especially important in scenarios such as collision detection and for motions that maintain or restore balance of the robot.

In this paper, we survey the existing controller boards to communicate with servos that utilize RS-485 (or TTL) for serial communication. Furthermore, we propose new approaches to improve the control loop rate using multi-bus approaches.

This paper is structured as follows: Section 2 introduces the communication with Dynamixel servo motors. In Section 3 we present existing controller boards. Section 4 explains two approaches. First, we show our improved firmware for an existing controller board. Second, we present our newly developed controller board. We evaluate the performance of controlling Dynamixel servo motors in Section 5. Afterward, the integration of sensors into the bus is discussed and evaluated in Section 6. The ROS [7] based software which interfaces with the controller boards is described in Section 7. A collection of lessons learned which might be helpful to anyone using the Dynamixel bus is provided in Section 8. Finally, the paper concludes with Section 9.

2 Dynamixel Communication

The Dynamixel bus uses a specified master/slave half-duplex protocol with 8 bit, 1 stop bit, and no parity [1]. All Dynamixel MX, X and Pro servos use either RS-485 or TTL as a physical communication layer. TTL communication can be emulated using RS-485 transceivers by tying the inverting data line to 2.5 V using a simple voltage divider. Since the minimum differential voltage on the RS-485 bus is 1.5 V, the 5 V TTL logic level is interpreted correctly. Each servo is assigned a unique ID number which is used to address the servo from the master.

There are two types of packages: instruction packages, sent by the master, and status packages, sent by the slaves. The master can send *write* instructions to set values to the slave's registers and *read* instructions to get current register values returned through a status package. The instruction packages can be either *single*, *sync*, or *bulk*. Single instructions are only processed by one slave. Sync packages specify a set of slaves and a range of registers with consecutive addresses which are read or written. All specified slave devices answer this instruction sequentially. Bulk packages are similar to sync packages, but it is possible to specify the registers for each slave individually. All packages start with a header, the corresponding ID, and the length of the package. The package ends with a checksum to verify its correctness.

Several features were introduced in the second version of the protocol, i.e. higher maximum package size, improve checksum and byte stuffing to prevent occurents of headers inside a package. Furthermore, version 2.0 specifies sync read and bulk write instruction packages which were not officially included previously. Since it is not compatible but superior to version 1, we are only considering the newer version of the protocol in the following.

read	sync read	bulk read
$i = n * 14$	$i = 14 + n$	$i = 10 + n * 5$
$s = n * (11 + r)$	$s = n * (11 + r)$	$s = n * (11 + r)$
write	sync write	bulk write
$i = n * (12 + r)$	$i = 14 + n * (r + 1)$	$i = 10 + n * (5 + r)$

i: Instruction length [B]
s: Status length [B]
n: Slaves to address
r: Registers to read/write

Table 1: Number of bytes needed for different package types.

Name	Microchip	Max. Bus Speed [MBaud]	No. Bus	Prot. 2	Cost
CM730[1]	FT232R+STM32F103	4.5	1	no	
OpenCM9.04	STM32F103	2.5 [2]	1	yes	50$
OpenCR	STM32F746	10	1	yes	180$
Arbotix Pro	FT232R+STM32F103	4.5	1	no	150$
Rhoban DXL	STM32F103	4.5 (2.25)[3]	3	yes[4]	∼20$
USB2DXL	FT232R	3	1	yes	50$
U2D2	FT232H	6	1	yes	50$
QUADDXL	FT4232H	12	4	yes	∼40$

Table 2: Comparison of multiple devices available for controlling RS485 or TTL servo Motors. The maximum bus speed describes the theoretically achievable maximum with the hardware. Some boards are limited in baud rate by firmware. [1] Discontinued; [2] Limited by transceiver on extension board; [3] Bus 1 can operate at 4.5 Mbps, bus 2 and 3 only at 2.25 Mbps; [4] Not supported by original firmware

While the protocol can be used for any device, it is most commonly used for the Dynamixel servos. They are a combination of a DC motor, a gearbox and a microprocessor which acts as a PID controller. The servo can be controlled by current, velocity, position or a combination of position and maximal current. In a typical closed-loop application the sensor values of the servos are read, and new commands are given to the servos in a fixed rate. The performance of the system depends thereby on how long a read and write cycle takes. Multiple factors influence the cycle time. These are the baud rate, the number of servos on the bus, and the number of bytes read and written. Table 1 gives the calculations for the required lengths of the packages required for *single*, *sync* and *bulk read* and *write* instructions. It is visible that the use of sync or bulk packages can decrease the number of needed bytes and therefore increase the performance.

3 Related Work

The Dynamixel bus system is widespread in different areas of robotics, mainly legged robots and robotic arms. Multiple robot platforms that use this bus system are produced by the manufacturer of the Dynamixel motors. These include the mobile robot TurtleBot3 [4], the small size humanoid robot DarwinOP [5], the full-size humanoid robot THORMANG, and the robotic arm OpenManipulator-X. Many research groups have also developed robot platforms that use this bus. They are especially widespread in the Humanoid League of the RoboCup where 32 of 34 teams use the Robotis servos and therefore the

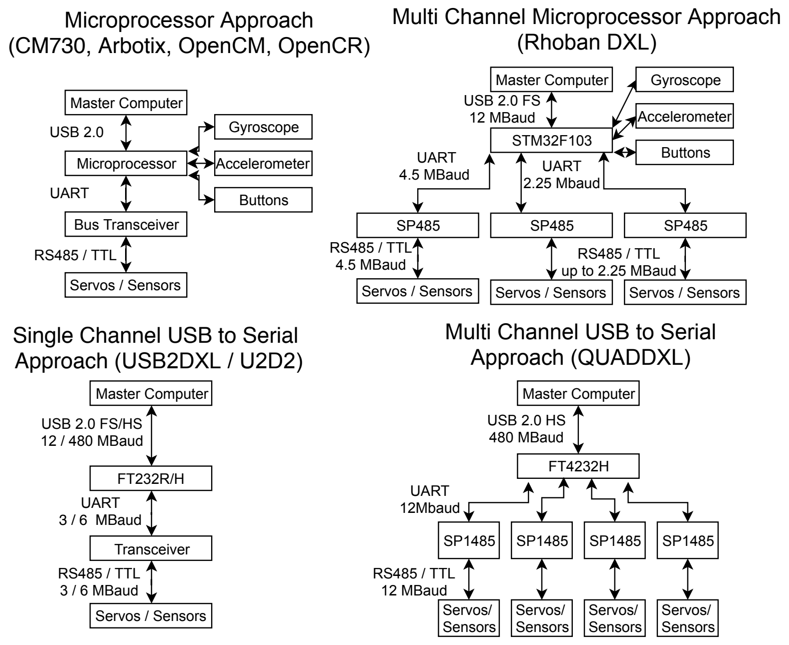

Fig. 1: Block diagram of different approaches to communicate with servo motors, sensors and other peripheral devices using the Dynamixel bus. They can be differentiated by the number of buses (**horizontal**) and the used chip type (**vertical**). Our newly proposed solution is the QUADDXL (**bottom right**).

Dynamixel bus system[2]. They are also used in multiple other leagues of the RoboCup, e.g., the Rescue League.

Since the actuators are used in many robot platforms, multiple controller solutions are available. Some of them are commercially available from Robotis or other distributors, e.g., the *Arbotix* controllers from Interbotix. Furthermore, there are multiple boards designed by research groups for their robotic platform, especially in the RoboCup domain. An overview of these boards is shown in Table 2.

All of these boards connect to a host computer via USB. They can be grouped into two different approaches. One approach uses a USB to serial converter to directly translate the USB signal to UART and afterward, by using a transceiver, to RS-485 or TTL. Two widespread representatives of this approach are the *USB2DXL* and *U2D2* boards from Robotis, which provide no direct sensor interface. The other approach uses a microprocessor which parses the packages coming from the host computer via USB. It then either transmits the packages via UART and an RS485 or TTL transceiver to the bus (if they are destined for a slave) or directly responds with a status package if values are requested

[2] Data from team description papers available at https://www.robocuphumanoid.org/hl-2019/teams/

from sensors (e.g., an IMU) connected to the processor. A block diagram of the different approaches is shown in Figure 1.
The only example of a controller which supports multiple communication buses to the servos is the *Rhoban DXL Board* [2]. Using multiple buses parallelizes the communication and thereby theoretically decrease the cycle time. Due to the limbed structure of humanoid robots, up to five separated buses are possible without increased cabling. A more detailed description of the performance of the approaches see Section 4.1.

In addition to servo motors, other peripheral devices such as sensors, buttons or displays can be attached to the Dynamixel bus. The AX-S1, a sensor module by Robotis, is one of these peripheral devices. Team Rhoban from the RoboCup Humanoid League developed a foot pressure sensor which can also be attached to the Dynamixel bus [2]. We present an improvement to Rhoban's device and a newly developed IMU module in Section 6.

4 Bus Controllers

Two different approaches were taken to improve the performance of the bus. First, we improved the well-performing Rhoban DXL Board by changing its firmware to adapt to protocol 2.0, enabling the use of sync commands. Second, a USB to four-channel serial chip was used to further increase the performance. The two approaches are presented in the following and evaluated in Section 5.

4.1 DXL Board

Since the DXL Board was the only available multi-bus controller, this board was chosen as the current baseline. It supports up to three buses. The original version had a firmware supporting only protocol 1.0 with a custom implementation of a sync read. The host computer can send a special sync read command to the board which translates it into regular read packages for the individual slave devices. Those instructions are then transmitted on the corresponding bus where the slave is connected. All status package are then aggregated and returned as a special response package. Thereby, the board can increase the throughput in comparison to a single bus, but it violates the protocol specifications.

Since protocol 2.0 specifies a sync read, there is no need for a custom sync read and it can be replaced by sending a standard sync package from host computer to the DXL board. This package is split up on the microprocessor of the board into three sync read packages for the three buses. The resulting status packages are transmitted to the host computer in correct order to stay in the protocol specifications. This way the number of required bytes on the servo bus can be further decreased and no extension of the protocol is necessary. The required bytes on the USB bus increase since single status packages are transmitted, but its influence is low since the USB bus has a higher throughput than the servo bus.

As a comparison, we implemented a second firmware version which only uses one bus system. Here, all instructions from the main computer are parsed and transmitted to the bus if it is not a reading of the directly connected sensors, i.e., the IMU or connected buttons. The resulting status packages do not have to be parsed and can be transmitted directly to the main computer. This reduces the workload on the microprocessor and the latency resulting from parsing packages. Furthermore, only one of the UART ports of the microprocessor is capable of 4.5 MBaud. The other two can only achieve a maximum speed of 2.25 MBaud. Therefore, this is the only way to utilize the highest possible baud rate of the Dynamixel servos.

4.2 QUADDXL

We developed a simple, high speed, low latency alternative to the DXL Board using the FT4232H chip from FTDI. It is a single USB 2.0 High Speed to 4 serial channel converter. Its schematic is presented in Figure 2. The virtual com port drivers required for using the device have been included in the Linux kernel since version 3.0.0, and the buses are directly accessible as serial device files. The interface is comparable to the USB2DXL and U2D2, but it offers four bus systems with up to 12 MBaud each while requiring only one USB connection. No firmware is required since the chip directly transmits the bytes without parsing packages. It is therefore also usable for both Dynamixel protocol versions and any future versions. Furthermore, the latency is lower since no parsing is involved.

5 Evaluation of Servo Communication

The typical control strategy of servo motors in a humanoid robot is a continuous cycle of reading their sensors' information and writing new goals. In contrast to other robot types (e.g., robotic arms or wheeled robots), continuous control is necessary to keep the robot from falling. A higher frequency of this cycle results in faster reaction time to disturbances, for example by correcting the torso pose after an applied force to keep the robot standing. Since using sync reads/writes is the most efficient way to achieve this (see Section 2), we focused our evaluation on these instructions.

In our experimental setup, we used 20 Dynamixel MX-64 servos using RS-485. 10 bytes are necessary to read the current position, velocity and used current from a single servo motor. The current is used to identify the torque applied by the servo. Additionally, it is necessary to write 4 bytes to each servo to set a goal position. If all servos are on the same bus, this results in 568 bytes of data per update cycle, including header and checksum bytes (see Table 1). In the case of splitting it up on four buses, the necessary data is 163 byte per bus, due to additional header bytes. We chose this 20 motor setup since it is similar to many humanoid robots, such as Darwin-OP[5] and most of the robots in the Humanoid League which were profoundly influenced by it. We measured the mean update cycle rate of our approaches together with the USB2DXL as a baseline and

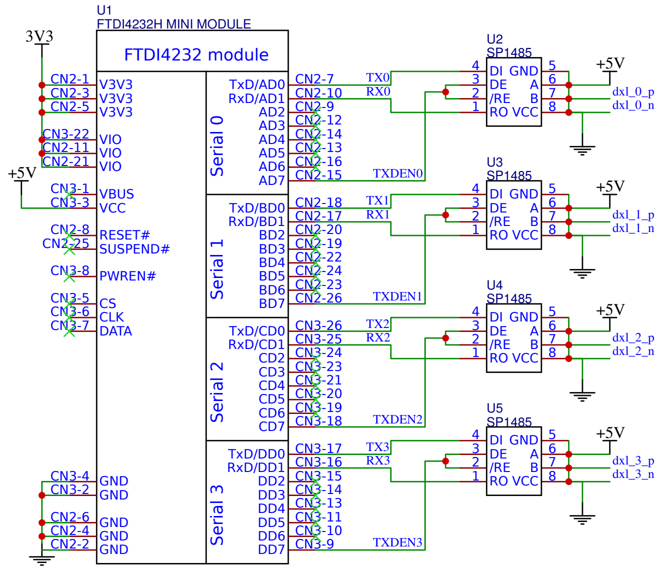

Fig. 2: Schematic of the QUADDXL. An FT4232H Mini Module (**left**) is connected to the host computer via USB. The four RS485 transceivers (**right**) are connected via the UART interface. The transceivers provide the required physical communication layer to a Dynamixel bus each.

additionally provided the theoretical maximum that could be achieved if the bus would be transmitting these bytes without downtime between packages. The results are displayed in Table 3 and a comparison between the QUADDXL and the theoretical maximum is displayed in Figure 3. We were not able to communicate with the servos on 4.5 MBaud with any controller. Therefore the highest tested baud rate is 4 MBaud. The reasons for this remain unknown.

For a single bus, the best performance was reached in 1 and 2 MBaud by the USB2DXL, closely followed by the QUADDXL. The Rhoban DXL performed worse in single and multi-bus approach. 4 MBaud is not possible with the USB2DXL but would have been with the U2D2. We expect it to perform slightly better than the QUADDXL in the single bus case (as the USB2DXL did on lower baud rates), but it was not available to us for testing.

Considering multiple buses, the Rhoban DXL performed worse than the QUADDXL. Furthermore, the performance of the Rhoban DXL board does not scale linearly with increasing bus number and baud rate. This indicates that the processing on the microprocessor is a bottleneck.

Overall, the QUADDXL manages to increase the cycle rate with increasing bus number and baud rate. It reaches a maximum of 1373 Hz which is three times higher than its speed on a single bus, and two times higher than the maximum possible speed on a single bus. The disparity between the real update rates and the theoretically possible rate is very high (see Figure 3).

No. buses	MBaud	Board	Rate [Hz]
1	1	Theoretical Max	176
		R-DXL Single	132
		R-DXL Multi	125
		USB2DXL	153
		QUADDXL	149
	2	Theoretical Max.	352
		R-DXL Single	215
		R-DXL Multi	179
		USB2DXL	272
		QUADDXL	261
	4	Theoretical Max.	704
		R-DXL Single	40
		QUADDXL	398
2	1	Theoretical Max.	336
		R-DXL Multi	185
		QUADDXL	285
	2	Theoretical Max.	671
		R-DXL Multi	219
		QUADDXL	497
	4	Theoretical Max.	1342
		QUADDXL	744
3	1	Theoretical Max.	461
		R-DXL Multi	224
		QUADDXL	390
	2	Theoretical Max.	922
		R-DXL Multi	250
		QUADDXL	670
	4	Theoretical Max.	1843
		QUADDXL	1003
4	1	Theoretical Max.	613
		QUADDXL	524
	2	Theoretical Max.	1227
		QUADDXL	923
	4	Theoretical Max.	2545
		QUADDXL	**1373**

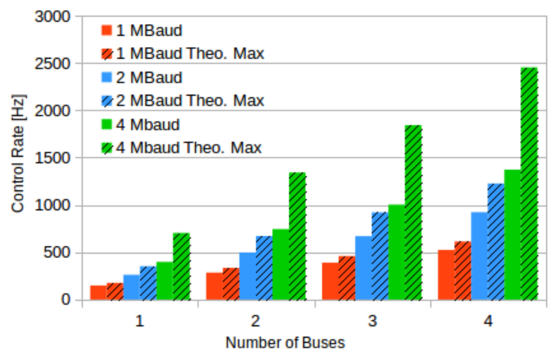

Fig. 3: Comparison of theoretically possible control rates with results from QUADDXL. The disparity between both is getting higher with increasing bus speed due to the constant response delay time.

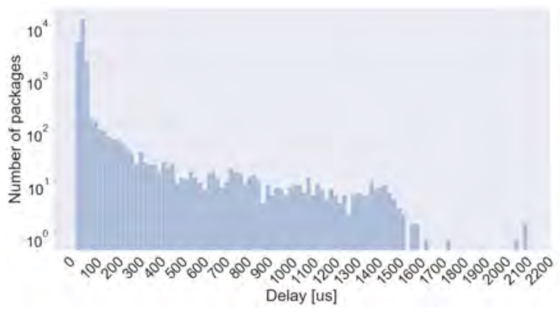

Fig. 4: Histogram of delays between status packages in a sync read response from Dynamixel MX-64 servos. The y-axis is scaled logarithmically. A total of 61,913 packages was analyzed. The bucket size for each bar is 20us.

Table 3: Mean update rates for different numbers of buses and baud rates with sync commands in protocol 2.0. The Rhoban DXL board (R-DXL) was tested with our newly programmed firmware in single- and multi-bus version. The QUADDXL approach provides the best results. We believe that the low rate for the R-DXL on 4MBaud is due to the microcontroller beeing to slow, resulting in massive package loss.

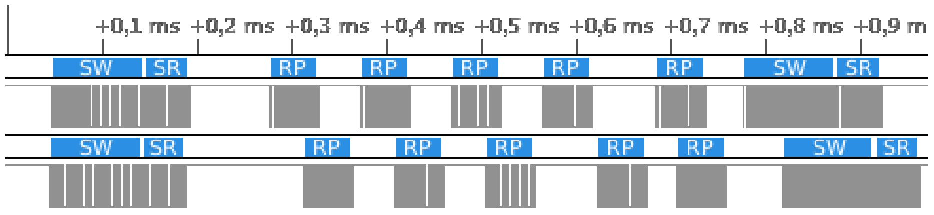

Fig. 5: Screenshot of a logic analyzer showing two buses with 5 servos each at 4 MBaud, which are controlled in parallel. The first package on the left is a *sync write* (**SW**), directly followed by a *sync read* (**SR**). These are followed by the response packages (**RP**) of the servos and then again followed by the *sync read* of the next cycle. The delay before response packages caused by the parsing of the packages is clearly visible.

To further investigate this significant difference, a logic analyzer was used. The results show that a significant portion of the available bus time is used up by the delay of the response packages from the servos (see Figure 5).
There is not only a delay between the sync read instruction and the first status but also between each of the status packages. This delay length does not change with a different baud rate. We assume that this delay derives from the servo's need to parse the previous package on the bus. In a sync read, the servo only sends its status package after it has read the status package of the servo which was specified before it in the instruction. This ensures that there are not two servos writing on the bus at the same time. This time varies as an analysis of the data recorded with the logic analyzer shows (Figure 4).
Since the firmware of the servos is closed source, there is no direct possibility to solve this problem. When comparing the performance influence of a higher baudd rate and more buses on a system with long response delays, more buses scale better than a higher baud rate (see also Figure 3). A higher baud rate can only decrease the bus time needed for transferring data but does not shorten the delays. More buses, on the other hand, can parallelize this delay.
The theoretically possible update rate for one bus can be computed using equation 1, where *data* is the number of bytes that have to be sent on the bus in each cycle (see Table 1). Since 10 bit are required to transmit 1 byte of data (1 start and 1 stop bit), this factor has to be added.
The optimal real rate, i.e., without possible delays introduced by the bus controller or main computer, can be approximated with equation 2 by introducing an additional factor that takes the response delay of the servo into account. In this equation, n is the number of servos on the bus and the $50\mu s$ are an approximation for the mean delay time.
While the experiments were conducted on MX-64 servos, we expect other motors from the MX and X series to behave identically since the same microprocessor (STM32F103) is used in all of them.

$$rate[Hz] = \frac{1}{\frac{data[B]*10[bit/B]}{baud[bit/s]}} \quad (1) \qquad rate[Hz] = \frac{1}{\frac{data[B]*10[bit/B]}{baud[bit/s]} + n * 50[\mu s]} \quad (2)$$

6 Sensors

Besides the sensor data from the servos themselves, additional modalities are often needed in the control cycle. In the RoboCup Humanoid League, those are mostly IMUs and foot pressure sensors. If we want to benefit from the higher control rate of the servos we also need a similarly high reading rate of the sensors. Different possibilities to connect sensors to the main computer exist. The most practical way is to include them on the Dynamixel bus since this reduces the number of required cables. Furthermore, it allows for interfacing the sensors with the same software as the servos.
We present and evaluate two different sensor boards that use the Dynamixel bus for communication to the main computer. Firstly, an IMU sensor module is introduced in Section 6.1. Secondly, a foot pressure sensor based on Rhoban's ForceFoot [2] is presented in Section 6.2.
While only these two boards are presented and evaluated, practically any sensor using common physical communication interfaces can be attached to the Dynamixel bus using a microprocessor. Furthermore, other peripheral electronics such as buttons and displays can be added to the bus system. The general approach is presented in the schematic shown in Figure 6.

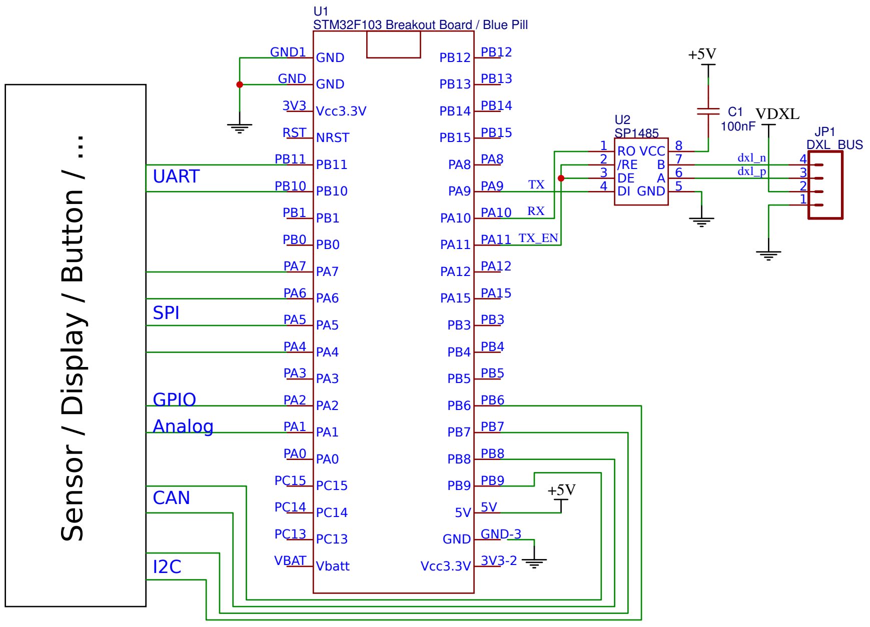

Fig. 6: Schematic for a generic sensor or IO module using the Dynamixel bus. An STM32F103 breakout board (**center**), e.g., a Blue Pill, connects via UART to an RS485 transceiver (**right**) which is connected to the Dynamixel bus. A multitude of peripheral devices (**left**) may be attached to the various communication interfaces provided by the microcontroller.

6.1 IMU

We used an MPU6050 IMU connected to a Maple Mini using I2C at 400KHz. The Maple Mini is connected to the Dynamixel Bus using an RS-485 transceiver. The IMU provides the raw linear accelerations and angular velocities in a resolution of two bytes each. Therefore it was necessary to read 12 bytes in each cycle on one device. Since this only requires a single read and status package, 37 Bytes needed to be transmitted (see Table 1). Due to the low number of bytes needed for this process, it was possible to reach an update cycle of 1kHz. The rate is limited by the fact, that the microprocessor has a single core can not read the IMU and communicate on the Dynamixel bus at the same time. Using a multi core microprocessor could solve this, if higher update rates are required.

6.2 Foot Pressure Sensor

The most widespread foot pressure sensor in the Humanoid League was developed by Rhoban [8] and consists of four strain gauges with an HX711 ADC and an ATMega328PB chip to connect it with the Dynamixel bus. While this approach proved successful, the employed ADC limits the update rate to 80Hz. Therefore, we developed an improved version, called *Bit Foot* (see Figure 7), using an ADS1262. This ADC can sample four analog signals with a resolution

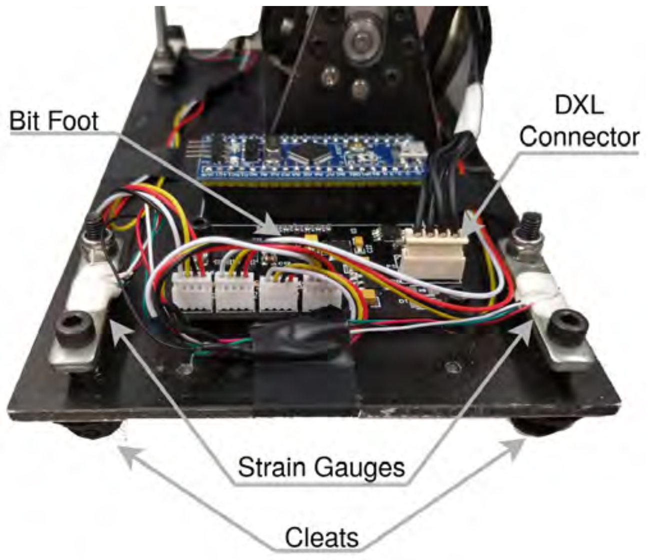

Fig. 7: The Bit Foot, a sensor module following the pattern described in Figure 6. Cleats are attached to strain gauges at four corners of the foot to measure the contact forces of the robot's foot. The analog signals of these strain gauges are fed into an ADC. SPI is used to interface this microchip from the STM32F103. The board is connected to the Dynamixel bus via an RS485 transceiver.

of 32 bit at a rate of up to 38400 Hz. It provides integrated filtering and multiple analog channels. It is connected via SPI to a STM32F103 breakout board called *Blue Pill*. This board is connected to the Dynamixel bus via an RS-485 transceiver. We use the raw data for all four strain gauges (32 bytes), but it would also be feasible to directly provide the center of pressure.
We evaluated the sensor similar to the IMU using regular read packages but only reached a cycle rate of 697 Hz. The readout of the ADC limits this rate. It can not read the four sensors at the same time but is only multiplexing between the different inputs. When changing the input channel, a delay time is needed before reading new values. This problem could be solved by using a different or multiple ADCs. Our result is still a significant improvement to the current baseline.

7 Interface Software

In order provide sensor and servo data to other software components, interfacing software is required. We chose the ros_control framework [3], which abstracts hardware and provides different controllers. Furthermore, it directly integrates the hardware into ROS [7] and provides the same interface when using the Gazebo simulator [6], simplifying the transfer from simulation to the real robot.
We implemented a hardware interface which can connect the servos and the previously mentioned sensors. The protocol implementation is based on the Robotis Dynamixel SDK[3] and Workbench[4] with the addition of a multi-register *sync read* which is necessary to read all values of the servos in a single *sync read*. The devices and values which should be read can be specified in a configuration file. Multiple buses can be used in parallel by initiating multiple instances of this interface, which is essential for the QUADDXL approach.
Different controllers provided by ros_control, e.g., position or effort, can be directly used with this hardware interface. Since the Dynamixel servos provide the possibility to use *Current-based Position Control*, where the servo is position controlled but with a maximum current, an additional controller was implemented to make this possible.

8 Lessons Learned

This section provides some important practical information which was learned during the work.
Sync Reads The use of *sync reads* and *writes* is essential to increase performance since they reduce the number of bytes needed for each control loop cycle. If registers are used which have no sequential addresses, the indirect addressing feature should be used to reduce the number of needed sync packages.

[3] http://emanual.robotis.com/docs/en/software/dynamixel/dynamixel_sdk/overview/
[4] http://emanual.robotis.com/docs/en/software/dynamixel/dynamixel_workbench/

Return Delay Time The Dynamixel servos have a *return delay time* value which adds an additional delay to the response time. The default value is 250 us, which can make fast communication impossible, even when this delay is set in just a single servo. At each startup of the robot, this register value should be written to 0, in case that a servo was replaced or reset, since difficult to debug bus timeouts and performance drops can happen.
Linux Kernel Latency The default value of the Linux USB-serial device latency timer is 16ms. This is done to reduce the number of needed headers for the USB packages and the load on the CPU. We set it to 0 to reach the best performance. Due to the high baud rate of USB 2.0 HS (480 MBaud) the additional header bytes do not slow down the sensor and servo communication.
Use Highest Baud Rate Possible Even if a fixed cycle time is used which would be achievable with a lower baud rate, the transfer time of a package is lowered, thus making the reaction of the servos faster. We did not encounter any problems with noise or corrupted packages on higher baud rates.
FTDI Bus Controller The latency of the USB to serial devices is lower than the microprocessor-based approaches since those have to parse the packages which introduces an additional delay. Furthermore, with the USB to serial approaches, no additional firmware is necessary, thus simplifying deployment and debugging.
Open Source Firmware It was not possible for us to improve the long reply delays of the Dynamixel servos. This problem could possibly be solved if the firmware would have been open sourced.

9 Conclusion

In this paper, we evaluated methods to improve the interfacing of servos and sensors using the widespread Dynamixel protocol. Our approach of using multiple buses showed to be the most effective due to the long reply times of the Dynamixel servos and their limitations in baud rate. Using this, we achieved a cycle rate of more than 1 KHz on the servos of our robot. Furthermore, we showed an approach to integrate other sensors with high update rates into the same bus, thus reducing the number of cables in the robot. The presented solutions are very low-cost and easy to reproduce.
Further improvements on cycle rates could be reached by reducing the response time of the servos or by extending the Dynamixel protocol to reduce the number of bytes per cycle. One approach would be a *cyclic read* and *cyclic write*, where the master specifies which values are written and read each cycle before starting. Afterward, only shortened request and response packages can be sent since all slaves know which data they need to transmit.
We invite other teams to use our controller board[5], control software[6], foot pressure sensors[7] and sensor connection board[8].

[5] https://github.com/bit-bots/bitbots_quaddxl
[6] https://github.com/bit-bots/bitbots_lowlevel/tree/master/bitbots_ros_control
[7] https://github.com/bit-bots/bit_foot
[8] https://github.com/bit-bots/dxl_sensor

Acknowledgments

Thanks to the Hamburg Bit-Bots for support. Thanks to team Rhoban for making their hardware and software open source. This research was partially funded by the German Research Foundation (DFG) and the National Science Foundation of China (NSFC) in project Crossmodal Learning, TRR-169.

References

1. Robotis e-manual. http://emanual.robotis.com/ (accessed 21.04.2019)
2. Allali, J., Deguillaume, L., Fabre, R., Gondry, L., Hofer, L., Ly, O., N'Guyen, S., Passault, G., Pirrone, A., Rouxel, Q.: Rhoban football club: Robocup humanoid kid-size 2016 champion team paper. In: Robot World Cup. Springer (2016)
3. Chitta, S., Marder-Eppstein, E., Meeussen, W., Pradeep, V., Rodríguez Tsouroukdissian, A., Bohren, J., Coleman, D., Magyar, B., Raiola, G., Lüdtke, M., Fernández Perdomo, E.: ros_control: A generic and simple control framework for ros. The Journal of Open Source Software (2017)
4. Guizzo, E., Ackerman, E.: The turtlebot3 teacher [resources_hands on]. IEEE Spectrum **54**(8), 19–20 (2017)
5. Ha, I., Tamura, Y., Asama, H., Han, J., Hong, D.W.: Development of open humanoid platform darwin-op. In: SICE annual conference 2011. pp. 2178–2181. IEEE (2011)
6. Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). vol. 3, pp. 2149–2154. IEEE (2004)
7. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA workshop on open source software. vol. 3, p. 5. Kobe, Japan (2009)
8. Rouxel, Q., Passault, G., Hofer, L., N'Guyen, S., Ly, O.: Rhoban hardware and software open source contributions for robocup humanoids. In: Proceedings of 10th Workshop on Humanoid Soccer Robots, IEEE-RAS Int. Conference on Humanoid Robots, Seoul, Korea (2015)
9. Soltero, M., Zhang, J., Cockril, C.: RS-422 and RS-485 Standards Overview and System Configurations (2002), Technical Report, Texas Instruments