**Proceedings of the ASME 2013 International Design Engineering Technical Conferences &
International Conference on Multibody Systems, Nonlinear Dynamics, and Control
IDETC/CIE 2013
August 4-7, 2013, Portland, Oregon, USA**

DETC2013-12493

OPEN-SOURCE REAL-TIME ROBOT OPERATION AND CONTROL SYSTEM FOR HIGHLY DYNAMIC, MODULAR MACHINES

Andrew Peekema*
Dynamic Robotics Laboratory
Oregon State University
Corvallis, Oregon 97331
Email: peekemaa@onid.orst.edu

Daniel Renjewski
Jonathan Hurst
Dynamic Robotics Laboratory
Oregon State University
Corvallis, Oregon, 97331
(daniel.renjewski,jonathan.hurst)@oregonstate.edu

ABSTRACT

The control system of a highly dynamic robot requires the ability to respond quickly to changes in the robot's state. This type of system is needed in varying fields such as dynamic locomotion, multicopter control, and human-robot interaction. Robots in these fields require software and hardware capable of hard real-time, high frequency control. In addition, the application outlined in this paper requires modular components, remote guidance, and mobile control. The described system integrates a computer on the robot for running a control algorithm, a bus for communicating with microcontrollers connected to sensors and actuators, and a remote user interface for interacting with the robot. Current commercial solutions can be expensive, and open source solutions are often time consuming. The key innovation described in this paper is the building of a control system from existing - mostly open source - components that can provide real-time, high frequency control of the robot. This paper covers the development of such a control system based on ROS, OROCOS, and EtherCAT, its implementation on a dynamic bipedal robot, and system performance test results.

INTRODUCTION

Dynamic locomotion requires a software and electronic system that can command signals based on sensor data in short time windows. While at first glance this may appear to be a unique application, it is only one in a rapidly growing field that demands hard real-time, high frequency control. Other applications include impedance controlled reaching [1], human-robot interaction [2], dynamic manipulation [3], dynamic image control [4], and multicopter control [5], to name a few.

Given the many applications, it would seem that hard real-time, high frequency control software and electronic structures should be easily available. Commercially available solutions are typically all-in-one bundles that cover electronics and software, however they are quite costly and closed source. Especially - but not only - in scientific projects, the use of commercial systems is constrained by limited budgets and the need to share and reproduce research findings. Open source solutions on the other hand offer free software, run on standard, low cost electronics, and facilitate concept reuse [6]; but they often take a large sacrifice of time and effort to integrate into a functional system. The described design is a combination of mainly open source components with an aim to reduce custom code.

The following is a comprehensive robot control system designed for, but not limited to, the control of a dynamic bipedal robot (ATRIAS, Figure 1) [7]. This paper describes the software and electronics - specifically system constraints, final design, and implementation. This system has been successfully deployed on the aforementioned bipedal robot.

*Address all correspondence to this author.

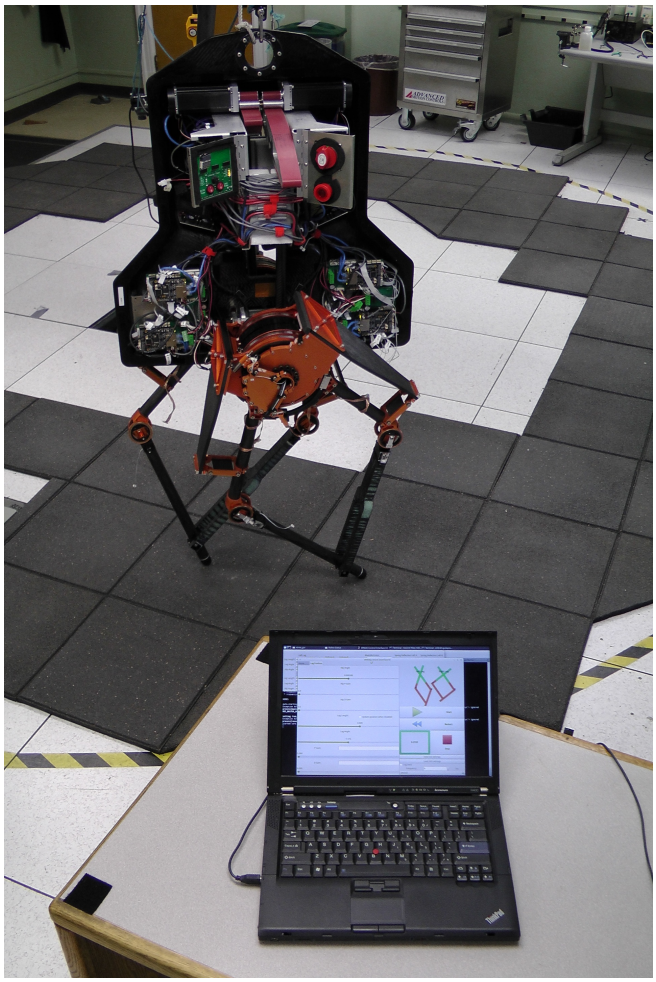

FIGURE 1. THE ROBOT (ATRIAS) AND GRAPHICAL USER INTERFACE.

SYSTEM REQUIREMENTS AND CONSTRAINTS

System architecture requirements can be broken down into three categories: physical, control, and software. The application is a robot that walks around, so the physical focus is mobility. Since the robot will undergo dynamic actions, the control requires determinism at a high execution frequency. Lastly, the software needs to minimize effort, cost, and maintenance, so well-supported open source components were chosen.

Physically, the control computer must be located on the robot and the user interface has to be a remote device. The robot consists of a number of physical modules (e.g. leg, body, boom) that are developed and built in parallel and equipped with different sensors and actuators. The electronics stacks should minimize unique components, support a variety of standard IO-protocols, easily interconnect with each other, allow for independent testing, and easily support the addition of extra sensors and actuators if necessary. Therefore the electronics stacks must be modular, use a bus system that allows for various topologies, and interface with the control computer using standard electronics.

The control software has to be deterministic and execute at a high frequency. Determinism means real-time processing, with no missed cycles and process priority scheduling. The execution frequency goal is 1 kHz, which is fast enough for controllers to use signal derivatives. The control cycle includes taking sensor samples, generating control output, and passing control output to motor amplifiers before the next cycle begins. The controller also must be a software component on the control computer (instead of microcontroller firmware) for ease of testing and debugging. Finally, controller input and output need to be consistent from controller to controller, so they can be easily swapped.

The software must be a combination of well supported open source software components to maximize code reuse and minimize development time. Because the bipedal locomotion control concepts [8, 9] are not necessarily developed in robot research labs, the software must have a simple controller implementation and interface to increase usability. All of the software needs to be documented, and provide drivers for the chosen physical and wireless communication systems. It is also necessary for the software to have a system dynamics simulator, so that controllers can be validated without hardware risk. Lastly, the software needs to be able to log controller and sensor data.

SYSTEM DESIGN

The driving factor for the control system design has been the required communication bus bandwidth and modular topology between the microcontrollers and control computer. Out of a number of available bus systems (Table 1), EtherCAT was chosen because it uses standard Ethernet electronics, has flexible network topology, and provides tight device synchronization. In comparison to other bus infrastructures, it can be implemented using standard Ethernet network cards supported by the open-source EtherCAT driver (RT Net) and low cost slave control electronics. For real-time robot control a number of commercial products are available including: LabVIEW, Simulink, dspace, and Gostai RTC. LabVIEW and dspace have dedicated real-time hardware provided by their respective companies. They allow for convenient software development and support many electronic components, but are pricy and cannot be freely shared between developers at different labs. Also - at the time of development - none of these products supported EtherCAT. Therefore the control system was designed using standard, off-the-shelf electronics and open source software packages. The developed software is freely available [1].

Given the mobile nature of the robot, a nettop (Mini PC) is the best balance of power consumption, size, and processing power

[1] http://code.google.com/p/atrias/source/checkout

TABLE 1. COMPARISON OF BUS SYSTEMS

Name	Description
EtherCAT	Ethernet wiring, requires dedicated slave hardware
PROFINET IRT	Ethernet wiring, requires dedicated master and slave hardware
CC-Link IE	Optical Ethernet wiring, ring topology, requires dedicated master and slave hardware
SERCOS III	Ethernet wiring, line/ring topology, requires dedicated slave hardware, optional dedicated master hardware decreases jitter
Powerlink	Ethernet wiring, requires dedicated slave hardware
CAN Bus	Controller Area Network Bus. Fastest standard rate: 1 Mbit/s [10]
Modbus/TCP	Ethernet wiring, requires dedicated slave hardware
Serial Port	Supported by most microcontrollers. Fastest standard rate supported by the XMEGA128: 115,200 Bit/s [11]

TABLE 2. SOFTWARE FRAMEWORKS

Name	Description
ROS	Drivers, libraries, data visualizers, communication system, and package management
YARP	Drivers, libraries, and communication system
Urbi	Drivers, software component generation, scripting language, IDE. Partially closed-source.
Player	Drivers, libraries, and communication system
Rock	Drivers, software component generation (real-time), scripting language (real-time), data visualizers, and communication system (real-time)
OROCOS Toolchain	Software component generation (real-time), scripting language (real-time), and communication system (real-time)

for the control computer on the robot. Another constraint is the network card, which has to be supported by the EtherCAT drivers. The selected nettop is an OEM Production i1000A-i5B1. The machine that runs the user interface can be any computer as long as it can run Linux, has a wireless card, and connect to a monitor, keyboard, and mouse.

Linux is a natural operating system choice because it is open source and all of the robotics software frameworks listed in Table 2 support it. Specifically, Xubuntu is the operating system on the robot's control computer, and was chosen because of its large community, variety of supported electronics, and lightweight desktop environment.

A software framework needed to be selected to facilitate wireless communication between the robot and GUI computer. A number of robotic software frameworks exist, and several of them could have been chosen for the non real-time components. ROS was selected because of its large and active community, interoperability with most other frameworks, and encouragement from the grant sponsor.

Another required software framework is for generating real-time software components, which enable hard real-time control. OROCOS was chosen as the real-time control environment because of its support for different kernels and ROS communication capability.

In order to achieve real-time in Linux, a real-time kernel is required. The main open source options are: RTAI, RTLinux, RT-Preempt, and Xenomai. RTLinux is no longer developed, and so was not used. RT-Preempt was tested, but was found to have insufficient real-time performance for this application. RTAI was tested next, but the EtherCAT library at the time (EtherLab) was not real-time safe in user space. It was real-time safe in kernel space, but this increased custom software complexity. Xenomai was chosen because of consistent response time under load [12], sufficient real-time performance, EtherCAT support, and ORO-

COS support.
The system also needs to have an emergency stop (E-Stop), which will cut power to the motors under certain conditions. This needs to be implemented at a low level (firmware on microcontrollers) and be able to override the current program at any time (hardware interrupt). It also needs to be implemented outside main communication channels for redundancy and firmware simplicity.

SYSTEM OPERATION

A user can load a controller on the control computer by clicking a button on the graphical user interface (GUI) running on a remote computer (see the system structure in Figure 2). The instruction signal is wirelessly transmitted to the control computer, and received by a non real-time state machine. This program determines if the request is valid (the controller is not already loaded), and if so requests permission to load a controller from a real-time state machine that handles controller input and output. Once granted, the non real-time state machine loads the controller and signals the real-time state machine that the controller loaded. The non real-time state machine then wirelessly signals to the GUI that the controller was successfully loaded. This path is also used by the GUI to stop, start, and reset controllers.
A control cycle starts with a clock pulse sent out from all of the bus microcontrollers ① to the sensor microcontrollers ② (Reference Figure 2 to follow ⓗ's in this paragraph). The sensor microcontrollers request cached controller commands from the bus microcontrollers and poll sensors, attaching timestamps to sensor data as it comes in. When all of the data is gathered, the sensor microcontrollers push sensor data to and pull controller commands from the bus microcontrollers. Subsequently, the sensor microcontrollers relay the controller commands to motor amplifiers ③. After a predetermined delay (300 microseconds), the bus program ④ on the control computer requests the cached sensor data from the bus microcontrollers. When the bus program receives the sensor data, it relays it to a real-time state machine ⑤ that handles controller input and output. This in turn relays the sensor data to the controller ⑥. After processing, the controller returns its output to the real-time state machine, where the controller inputs and outputs are sent to a non real-time process to be logged ⑦. The controller outputs are then relayed to the bus program, which sends it out to the bus microcontrollers. The commands wait there until the next clock pulse.

SYSTEM IMPLEMENTATION

Recall that Figure 2 is a general overview of the system's components and their interconnection. It shows how the user interface, control components, and electronics communicate. OROCOS and EtherCAT handle real-time communication, while ROS handles non real-time communication. The main components are described in detail in the following sections.

Xenomai

Xenomai provides a real-time environment out of the box with its patch to the Linux kernel and corresponding libraries. A key feature of Xenomai is its ability to determine if a process switches between real-time in Xenomai (primary mode) and non real-time in Linux (secondary mode). This process is called mode switching. The /proc/xenomai/stat file lists all of the tasks within Xenomai and how many times they have mode switched (along with other statistics). Seeing how the mode switch count of a process changes over time is critical for understanding real-time bugs.

OROCOS

The only modification necessary for OROCOS to use Xenomai bindings is to set a shell environment variable. All controller components run within the OROCOS deployer, which allows the code to be kernel-agnostic. Code can be written and tested on a standard Linux kernel, and then verified for real-time performance on a Xenomai kernel. This enables productive code development on a standard Ubuntu kernel with less effort than a full Xenomai install.

Controllers

Controller components are implemented in a hierarchical style that enables code reuse and reduces bugs. For example, a PD controller can be implemented once. Another controller can instantiate several copies of the PD controller and create higher-level functionality - such as a startup controller. This is just one simple example; there is no nesting depth limit.
Controller input and output are standard C++ structs used by all controllers, so controllers can be swapped without the rest of the system having to change. This also standardizes log files, because these two structs are always logged.
Controllers can be tested in place using the Gazebo simulator. In order to accomplish this, the EtherCAT Connector is replaced by a Simulation Connector. This program acts identically to the EtherCAT Connector as far as the controller is aware, but instead of sending and receiving signals from the robot everything takes place in Gazebo. The Simulation Connector uses RTT-ROS Integration to communicate with a Gazebo plugin, which handles translating the physics simulation into the C++ controller input struct.

Distributed Clock, EtherCAT, and Microcontrollers

The purpose of the distributed clock is to keep all of the microcontrollers in sync with each other and the control

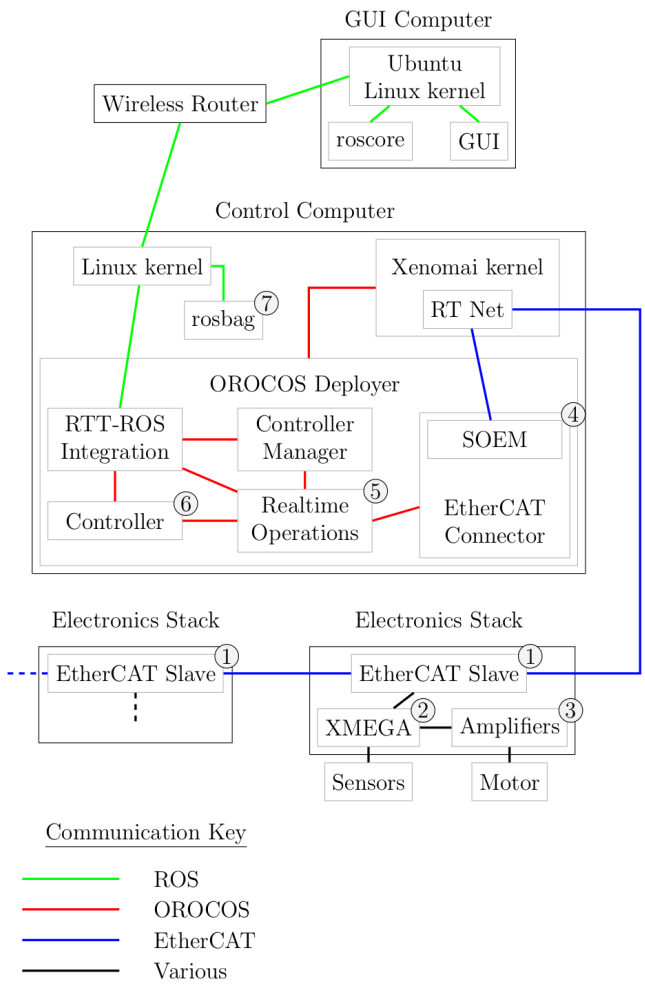

FIGURE 2. THE ROBOT SOFTWARE STRUCTURE

computer. The first distributed clock capable EtherCAT slave is the reference clock for the system. When the EtherCAT master starts, it sends out a series of packets to determine the time delay between each of the slaves. During run time the EtherCAT master attaches a command to an EtherCAT frame that causes the reference clock to put its time into that frame. All other EtherCAT slaves use that time to synchronize their own clocks, using the delays determined on startup. This is accomplished through several functions in the SOEM (Simple Open EtherCAT Master) library, with RT Net as the EtherCAT driver backend. The synchronization between slaves can be highly accurate; in practice, the difference between slave clocks has been around 30 nanoseconds. To put this in perspective, this is approximately equal to the time taken by one instruction cycle of the XMEGA microcontrollers being used (clocked at 32 MHz).

The EtherCAT slave chips are configured to generate a signal every millisecond, based on the distributed clock time. This signal is sent from the EtherCAT slave chip to a microcontroller pin with a hardware interrupt attached. The microcontroller then to reads sensors (Figure 3) and attaches timestamps to data as it comes in. Once complete, the microcontroller writes sensor data to, and read current commands from, the EtherCAT slave chip. The last step the microcontroller takes is to set PWM outputs that signal the amplifiers how much current to command. 300 microseconds after the distributed clock pulse occurs, the EtherCAT master sends out a frame to read the data from the EtherCAT slave chips. The controller is run using this data, and the EtherCAT master writes a frame containing the controller output to the EtherCAT slave chips. This process loops continually until the microcontrollers leave the run state.

GUI

The GUI is divided up into generic and controller specific portions. The generic section is constantly displayed, and shows data about the robot state (motor positions, current commands, and error states) as well as an interface to start, stop, and reset the currently loaded controller. The controller specific section is displayed only when the controller is loaded, and allows the controller writer to display information from and send information to the robot. The GUI is based on a plugin system written in GTK Builder for the interface, and shared libraries (written in C) for functions that are called by the interface. The GUI computer is connected to the control computer with a wireless router, and uses ROS's publish-subscribe system along with OROCOS's RTT-ROS integration to communicate with the controller and controller manager.

Logging

Rosbag, a datalogging tool provided by ROS, is used to log all GUI and controller data in a binary '.bag' file format. By using the same publish-subscribe system mentioned above, rosbag can easily capture all data over multiple machines as long as the data exists within the ROS network (connected to the same ROS master).

In order to capture data generated within OROCOS, OROCOS's RTT-ROS integration capabilities are used to push controller output, sensor data, system status, and controller-specific messages out of OROCOS and into ROS to be logged by rosbag. Rxbag, another tool provided by ROS, allows for simple online plotting of numerical data from rosbag log files. A python script exports rosbag files to MATLAB format.

Emergency Stop Bus

The Emergency Stop (E-Stop) bus is a daisy-chained set of wires between the microcontrollers and an external E-Stop

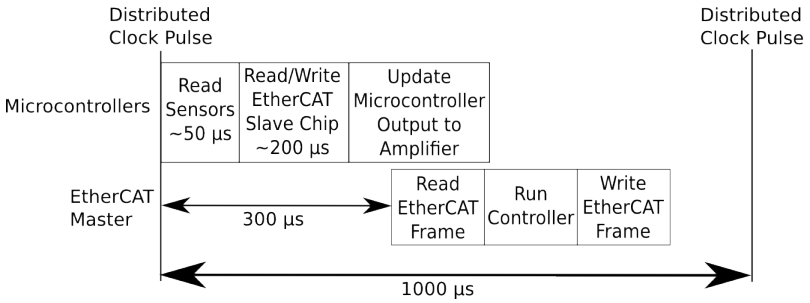

FIGURE 3. DISTRIBUTED CLOCK TIMING.

switch. The E-Stop line is triggered when the line is pulled low, so if there is a disconnection (ex. the E-Stop switch is pressed) it causes all of the microcontrollers to be sent into E-Stop. Each microcontroller has a dedicated e-stop input connected to a hardware interrupt, which triggers the E-Stop state. It also has a dedicated output line used to assert the bus.

The conditions for an emergency stop are outlined in Table 3. When a microcontroller receives any of these signals the E-Stop output line is asserted, and the motors attached to that microcontroller are disabled (The one exception is before a hard stop impact. In this case the motor actively stops its rotation for 100 milliseconds before asserting the E-Stop output line and disabling the motor). When the EtherCAT master sends a reset command the microcontroller de-asserts its E-Stop output line, clears any internal error states, and transitions into the idle state. In the idle state it ignores the E-Stop line - to prevent looping back into E-Stop - and waits for the run command.

SYSTEM EVALUATION

A variety of performance tests can be designed to demonstrate the capabilities of software systems. Two tests have been designed to document the timeliness and safety of the system.

Jitter

Jitter is the deviation of the control loop timing from the desired control frequency. Different scenarios were tested: 1) system idle, 2) controller running, 3) high network load and 4) high CPU load. For each control cycle, the desired and effective start time (based on the EtherCAT reference clock) is recorded. Each scenario ran for five minutes, as this is the time range of usual experiments. The results are seen in Figure 4. For the purpose of processing sensor data and generating motor commands to control a bipedal robot, the observed performance is acceptable by all means.

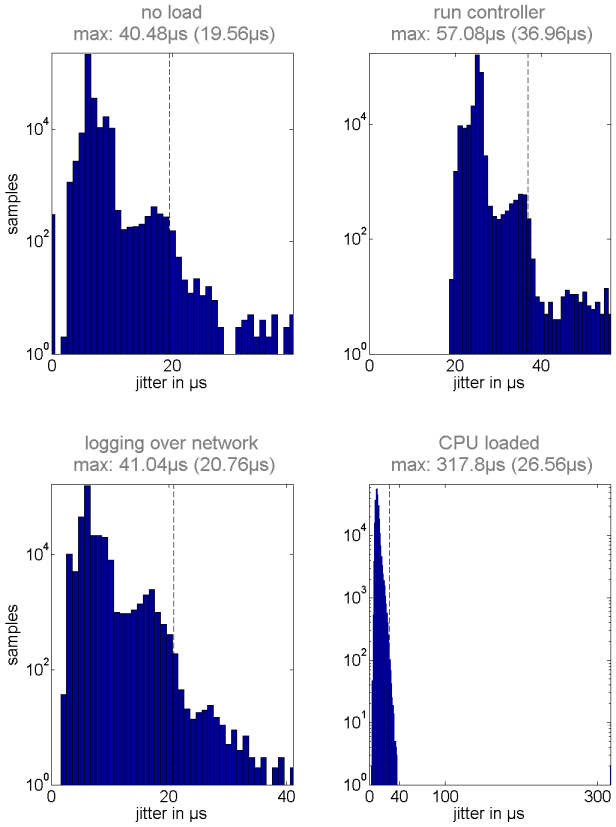

FIGURE 4. HISTOGRAMS FOR JITTER AT DIFFERENT LOAD CONDITIONS. DASHED LINES MARK THE MAXIMUM TIME DELAY FOR 99.9% OF THE SAMPLES (ALSO IN PARENTHESIS IN EACH FIGURE TITLE).

Emergency Stop

As safety is an important issue, the system's response to an emergency stop was measured. All processing is done on

TABLE 3. MICROCONTROLLER EMERGENCY STOP CONDITIONS: AN EMERGENCY STOP IS TRIGGERED IF ANY OF THESE SIGNALS OCCUR.

Description	Signal
Something is wrong with another microcontroller or the E-Stop wiring	The E-Stop line is low
A motor is going to hit a hard stop	Projected motor position is going to impact a hard stop given motor acceleration limitations
The robot hits a hard stop	A limit switch is pressed
EtherCAT communication is not reliable	5 milliseconds without an EtherCAT packet
The battery is low	Low motor or logic voltage
A motor temperature is too high	Low thermistor resistance

TABLE 4. TIME FROM INITIALIZING AN EMERGENCY STOP UNTIL MOTOR COMMAND TO STOP THE MOTORS IS ISSUED.

event	manual	limit switch
duration	0.7μs	3.3μs

the slave microcontrollers and the emergency stop signal is distributed in parallel to the EtherCAT bus. For this test, a continuous current signal was sent to the amplifiers. Then, the response time from triggering an e-stop until the amplifier cut the motor current was measured using a digital oscilloscope. Two scenarios were considered: 1) manually pressing an emergency stop button and 2) triggering a limit switch (Table 4). The system shuts down well within one control cycle when the e-stop button is pressed. Filtering requirements for the limit switch signals to remove electrical noise leads to a larger but uncritical delay until the motors are shut down.

CONCLUSION AND FUTURE WORK

This paper presents the successful implementation of a low-cost, reliable, and open-source real-time robot control system. The bus system used allows for modular integration of sensors and actuators in flexible topologies. The control system can be reproduced on standard hardware, uses strongly supported open-source components, and requires minimal custom code.
The overall response time of the system can be improved by reducing the time it takes to communicate with the amplifiers. Current work includes designing hardware that allows an amplifier to directly communicate with an EtherCAT slave chip. Also, controller code clarity can be improved with a standard high level state machine, such that logic flow is easily visible. Existing controllers have state machines with varying degrees of readability. Lastly, controller verification can be improved with lock-step simulation communication - the implementation is functional but has detectable response delays.
The system in its current state has been successfully used for experiments in robot walking, and enables highly dynamic robot locomotion.

ACKNOWLEDGMENT

This project was funded by DARPA grant number W91CRB-11-1-0002 to J. Hurst. The authors would like to thank Kit Morton, Johnathan van Why, Michael Anderson, and Soo-Hyun Yoo for their contributions and feedback.

REFERENCES

[1] Chen, Z., Lii, N. Y., Wimböeck, T., Fan, S., and Liu, H., 2011. "Experimental evaluation of cartesian and joint impedance control with adaptive friction compensation for the dexterous robot hand dlr-hit ii". *International Journal of Humanoid Robotics,* ***08***(04), pp. 649–671.

[2] De Santis, A., Siciliano, B., De Luca, A., and Bicchi, A., 2008. "An atlas of physical human–robot interaction". *Mechanism and Machine Theory,* ***43***(3), pp. 253–270.

[3] Butterfaß, J., Grebenstein, M., Liu, H., and Hirzinger, G., 2001. "DLR-hand II: Next generation of a dextrous robot hand". In Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on, Vol. 1, IEEE, pp. 109–114.

[4] Okumura, K., Oku, H., and Ishikawa, M., 2012. "Lumipen: Projection-based mixed reality for dynamic objects". In

Multimedia and Expo (ICME), IEEE International Conference on, IEEE, pp. 699–704.

[5] Meyer, J., and Strobel, A., 2010. "A flexible real-time control system for autonomous vehicles". *41st International Symposium on Robotics (ISR) and 6th German Conference on Robotics (ROBOTIK)*, June, pp. 1 –8.

[6] Fitzpatrick, P., Metta, G., and Natale, L., 2008. "Towards long-lived robot genes". *Robotics and Autonomous Systems,* ***56***(1), pp. 29 – 45.

[7] Grimes, J. A., and Hurst, J. W., 2012. "The design of atrias 1.0 a unique monopod, hopping robot". In International Conference on Climbing and Walking Robots.

[8] Hubicki, C. M., and Hurst, J. W., 2012. "Running on soft ground: Simple, energy-optimal disturbance rejection". In International Conference on Climbing and Walking Robots (CLAWAR).

[9] Ernst, M., Geyer, H., and Blickhan, R., 2009. *Spring-legged locomotion on uneven ground: a control approach to keep the running speed constant*. ch. 78, pp. 639–644.

[10] Standard, I., 1993. "Iso 11898, 1993". *Road Vehicles, Interchange of Digital InformationController Area Network (CAN) for High Speed Communications*.

[11] Atmel. Atxmega128a1-au datasheet. [Retrieved 1/22/13].

[12] Brown, J., and Martin, B., [Retrieved 1/22/13]. How fast is fast enough? choosing between xenomai and linux for real-time applications. Tech. rep., Open Source Automation Development Lab.