A Distributed Robot Control Architecture Using RTAI

Diego Caberlon Santini and Walter Fetter Lages
Federal University of Rio Grande do Sul, Electrical Engineering Department
Av. Osvaldo Aranha, 103, 90035-190 Porto Alegre, RS, BRAZIL
diegos@ece.ufrgs.br, w.fetter@ieee.org

Abstract

In this paper, we present a distributed system to fill the need for an open architecture for robot control. The system is based on simple processing nodes called Actuator Interface Cards (AIC). The hardware is composed by an AIC for each joint of the robot. Each AIC contains a dspPIC processor and can drive a D.C. motor with nominal voltage ranging from 12V to 60V through a PWM converter. Also, it interfaces with an incremental encoder performing quadrature decoding. A CANbus is used for real-time data transfer such as sensor measurements and actuator commands, between each AIC and a host PC. A real-time library running in the host PC, models the devices embedded in each AIC and supports all the possible operations, for example: shutoff and voltage setting for the motors and reading of count for encoders. This library also implements the communication with AICs using a protocol where a priority is associated with each command and card, thus ensuring the real-time requirements. The library accesses the AICs through CANbus with a real-time driver. A simple PID controller was implemented as a hard real-time thread in user-space using RTAI/LXRT modules to demonstrate the use of AIC and the real-time performance. Currently, an integration of the proposed architecture with the OROCOS project is under development.

1 Introduction

Robot controllers can be classified into three main categories: proprietary (also called closed), open and hybrid [2]. In a proprietary system, is very difficult, if not impossible, to integrate external components like sensors and software that aren’t supported by the robot manufacturer. But in an open system, anyone is able to add and change new components according to their needs. This is possible just because it is fully described. A hybrid system is a middle term between proprietary and open systems. Certain aspects such as the control law are closed whereas most elements are accessible.

Actually, proprietary systems have been used with most robotic manipulator. This cause problems if the vendor go out of business, or if spare parts can no longer be obtained. In this case, retrofitting of robots like [4] is a good way. However, this leads to a wide reformulation, which include working parts and may be very costly. With an open architecture, this problem can be better resolved by replacing only defectives components with another component with same value.

The main idea of an open architecture is the following [2]:

- Use a development system based on a non-proprietary computer platform. (e.g. SUN, SGI, PC’s);
- Use a standard operating system and a standard control language (e.g. C or C++);
- Base the hardware on a standard bus architecture that can interface with a variety of peripherals and sensor devices;
- Utilize networking strategies that allow work-cell controllers to share databases, and to be operated from remote locations.

Many of this characteristics can be found natively on RTAI/Linux [6], [5]: a development system based on PC with a UNIX-like operation system like RTAI/Linux. The module LXRT brings real-time capabilities in user space, allowing the program to use C++. Adding a hardware feature that works with RTAI/Linux and can handle a variety of manipulators robots will create a base for a distributed system to fill the need for an open architecture for robot control.
2 The Hardware Architecture

A scheme of the connection between the host PC and the AICs nodes is shown in Fig 1, where each AIC is routed in a standard Eurocard form-factor and can be viewed in Fig 2.

The hardware is composed by an AIC for each joint of the robot. Each AIC contains a dspPIC processor and can drive a D.C. motor with nominal voltage ranging from 12V to 60V through a PWM converter. Also, it interfaces with an incremental encoder performing quadrature decoding for sensing and a electro-mechanical brake.

The AIC has a dual function on this system. It can works as a actuator card, only making the interface between the hardware and the host PC, in this case, the control system is running on the host PC. In the other hand, it can also work as an intelligent card running the control algorithm. Although, due to its limited computational power, the AIC can not perform real-time control of complex nonlinear dynamic systems, it can handle the typical industrial approach where each joint of the robot arm is considered as an independent joint [3].

3 The Software Structure

The software structure can be divided in the four parts: the first one is the group of functions that supports the AIC hardware on dsPIC processor; after that came the protocol which realizes the communication between AICs and the host PC; using this protocol we have a hardware abstraction for AIC on the host; finally a real-time driver to access CANbus was made. All this parts are explained in next.

3.1 AIC Low Level Controller

A library was developed to provide a group of functions that runs on dsPIC processor and interacts with the hardware present OK AIC card. This functions supports all the possible operations on AIC, for example: off() and set(double voltage) for the Motors and read() for Encoders. This library also implements the communication for AICs through the CAN controller of dsPIC.

With the aim of executing the commands of host, a daemon was implemented. This task runs when the AIC is turned on. It starts the CAN controller and keeps polling for a CAN message. When a message come from host, the daemon interprets it and executes the corresponding command, according to Table 1.
3.2 Protocol AIC - Host

CAN-Bus is a broadcast bus with differential serial data transmission. The frames (show in Fig 3) consist of an ID to identify the message type and up to 8 data bytes is sensed from all CAN-bus nodes. A CAN network can be configured to work with different frame formats: the standard frame format has a 11Bits ID (CAN 2.0A) whereas the extended frame format uses 29Bits ID (CAN 2.0B). In this work, we work with CAN2.0A to save some time while sending receive commands.

Every bit transmitted on the bus is defined as recessive (1) or dominant (0). All nodes can listen and transmit at the same time. If more than one node is transmitting, the result will carry a dominant bit if at least one node is transmitting a dominant bit. If a node transmits a recessive bit, but a dominant bit is seen on the bus, the node knows that someone else is on the bus and stops his transmission. Hence, the priority of messages is defined by the ID field. With this in mind, we used the ID field to represent the command and to address one AIC. The Table 1 shows this relation.

For each message, a priority is associated with each command and card, thus ensuring the real-time requirements. It is easy to see that the high priority commands are safety commands, like turn off. This ensure, that in case of two commands like turn on motor and turn off motor are sent at same time, the turn off command will be transmitted first.

Also, if the same command is send for two AICs, the AIC which has the lowest ID will receive the command first, thus allowing the assignment of a priority for each joint of robot.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10-5</td>
<td>Command</td>
</tr>
<tr>
<td>000000 = Reserved</td>
<td></td>
</tr>
<tr>
<td>000001 = Reset</td>
<td></td>
</tr>
<tr>
<td>000010 = Turn off motor</td>
<td></td>
</tr>
<tr>
<td>000011 = Apply brake</td>
<td></td>
</tr>
<tr>
<td>010000 = Set voltage</td>
<td></td>
</tr>
<tr>
<td>100000 = Status</td>
<td></td>
</tr>
<tr>
<td>110000 = Release brake</td>
<td></td>
</tr>
<tr>
<td>110001 = Turn on motor</td>
<td></td>
</tr>
<tr>
<td>4-0</td>
<td>AIC address</td>
</tr>
<tr>
<td>00000 Reserved</td>
<td></td>
</tr>
<tr>
<td>00001 AIC 1</td>
<td></td>
</tr>
<tr>
<td>00010 AIC 2</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>11111 AIC 31</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 1: Description of ID field.

All commands have the direction from host to AIC, although the status command is a remote frame. The remote frame request that AIC send a standard frame with the status of AIC. All commands also are composed only of the ID field, except the command set voltage, and the answer of AIC status. This last two commands had the voltage that must be applied to motor, and the value of encoder. They are encapsulated with IEEE-754 single precision.

3.3 AIC Hardware Abstraction

An AIC hardware abstraction running in the host PC is needed to model the devices embedded in each AIC. In this work a real-time library in C++ was made to do this job. It is done using the concepts
showed in [7] like object-oriented programming such as encapsulation, inheritance, modularity, and polymorphism.

This is be done with the creation of AIC_COMM class. It defines the interface of communication. The interface establishes what requests you can make for AIC_COMM. This class does not implement any protocol, it just defines the interfaces with virtual functions.

A group of classes was developed to represent every device in AIC. These classes does not implement any action on the hardware, but they send commands (shown in Table 1) by using the AIC_COMM interface. After that, all objects of the former classes are declared in a new class AIC which models all AIC.

A final class AIC_CAN is derived from AIC_COMM and AIC and implemented the protocol over CANbus. The Fig 4 shows the inheritance diagram for hardware abstraction.

![Inheritance diagram](image)

**FIGURE 4:** Inheritance diagram.

Another benefit of the language C++ is its function and operator overload, which enables the definition of a interface very close to the problem. Here is a example for use AIC on a host.

```c
#include <aiccan.h>

int main(int argc,char *argv[])
{
    AIC aic=new AIC_CAN(1);
    float position;
    aic->brake.release();
    aic->motor.on();
    aic->motor=7.0;
    position=aic->encoder.read();
    //
    //

    3.4 Real-Time Driver for PCIcan-D

To ensure the real-time requirements a real-time driver was made for PCIcan-D card. This avoids the hard/soft real-time transitions that occurs on a normal Linux driver.

The driver access a SJA100 CAN controller over a PCI bus, interacting with user-space though kxrt module, like serial driver from source of RTAI. In the future this driver will be ported to RTDM interface.

4 Experimental Tests

In order to check the timing of the proposed system, the software was instrumented to set an reset bits at the parallel port of the PC. Thus it was possible to obtain real-time measures with help of oscilloscope. The Fig 5 show this signals along one cycle of the control loop.

![Timing of a control cycle](image)

**FIGURE 5:** Timing of a control cycle.

The pink and yellow signals are on host and green and blue are on AIC. At the left side, each blue spike shows a polling of the CAN controller on AIC. The first rise of yellow signal is a message send to the real-time driver on PC. The pink signal rises just before the transmit-ion of a message from the real-time driver on PC to AIC. At the end of message transmission, pink and yellow signals go down. This phase takes 50µs which is the time required to transmit the 44 bits of a remote command status frame (see Fig 3) at frequency of 1MHz plus some time that CAN controller introduces. Thereafter, the pink signal starts to spike indicating that host is polling CAN controller waiting a answer from AIC.

The reception of the message by AIC is signaled by the rise of the blue signal. The AIC receive the message and takes 42µs to prepare the status frame. The rise of the green signal corresponds to the AIC starting to send the status back to host. This is a total of 108 bits and takes 123µs. After that, yellow and pink signals rise showing that the driver is reading data from the CAN controller and as they go down, the control law computation starts. Then the yellow signal goes up again as the the voltage set command goes to the driver. The pink signal goes up indicating that the command is being sent. Again, this is a total of 108 bits and takes 116µs. Finally, the blue signal goes up as the AIC receives the command and
remains high until its processed, then all tasks sleeps and waits for a next loop.
The sum of those times (331µs) is a very important figure because it is the lowest period that the closed loop control can be performed.

5 Experimental Closed Loop Control

The software architecture described in the previous Sections has been used for a closed loop control of a robot joint. The control was done by a PID and more details about the robot control algorithm are not reported here for brevity and because is not the focus of this work.
The control is based in two real-time threads. One thread is hard-real-time and implements the PID using the AIC hardware abstraction with a 1ms as period. The other is a soft-real-time and save the data in log files. Both threads change data using mailbox from RTAI.
This PID receives like input a sinusoidal reference and the joint of robot presents a output as show in Fig 6.

![Reference × Output](image)

FIGURE 6: Reference × Output.

6 Conclusions

In this paper, the implementation of a distributed robot control architecture using RTAI has been presented. The experimental results show the effectiveness of the proposed control platform. This interface gives to the final user the possibility of implement controllers for robots.
In future developments, new models of hardware with other sensors and actuators like current sensors and servo motors will be developed. Currently, an integration of the proposed architecture with the OROCOS project is under development trying to create a fully open architecture for robot control.

References