Proc. 5th International Symposium on Methods and Models in Automation and Robotics MMAR'98, 25–29 August 1998, Międzyzdroje, Poland, Vol.3, pp.795–800

MULTI-ROBOT SYSTEM CONTROLLERS[†]

C. ZIELIŃSKI[*], A. RYDZEWSKI[*], W. SZYNKIEWICZ[*]

[*]Warsaw University of Technology, Institute of Control and Computation Engineering, ul. Nowowiejska 15/19, 00-665 Warsaw, POLAND, C.Zielinski@ia.pw.edu.pl

Abstract. The paper proposes a structure for open, hierarchical, multi-device controllers. The proposed structure takes into account that the system may contain several robots of different type, a certain number of cooperating devices, diverse sensors and the fact that the task, the system has to execute, and the number and type of its components may vary considerably over time. Both the hardware and software parts of the system are described herein. The flexibility of the system is due to the software, so the programming aspect is treated comprehensively in the paper.

Key Words. robot controllers, multi-robot systems, robot programming

1 INTRODUCTION

Robot controllers and the programming languages they interpret are inseparably bound together. Robots have to execute ever more complex and diverse tasks. The components of the system, i.e. number and type of robots, number and type of cooperating devices, number and kind of external sensors, that are necessary to carry out the job are not known before the task is specified and the solution to the problem is found. Controllers and programming methods of such systems have to take into account this fact. There are two solutions to this problem. Either the controller, and so its method of programing, has to be very universal, or the controller can be very specialised, but then it must be very easy to design, so that a specific controller for any task at hand can be designed quickly. This proposal follows the latter approach. It consists of: a general structure of the controller, a moderately sized set of construction modules that facilitate the construction of specialised controllers, and a method for both designing these controllers and for constructing and adding new modules to the original set.

Initially the idea of universal robot controllers and programming languages prevailed in the robotics community (e.g. `WAVE` [12], `AL` [11], `AML` [13], `RAPT` [1], `SRL` [5], `TORBOL` [14]). It soon turned out that such a controller has to be able to interpret a very complex language, having the same abilities that universal programming languages have and, moreover, extra capabilities for dealing with robots and sensors. Even when a general purpose programming language was adopted as a basis, a robot programming language had to have extensions (i.e. instructions and data types) due to specific devices composing the system. It was extremely difficult to decide what kind of additional components should this general purpose language contain so that any foreseen system could be controlled and programmed. So this "universal" approach has been given up for cases where it was expected that the system hardware may vary considerably with changing tasks. Paradoxically it turned out that the "universal" systems are much better suited to dealing only with initially well defined classes of tasks. In such a case the language can be tailored to the system configuration and the class of tasks at hand. Obviously, the broader the class the more universal the language.

Soon however, another idea emerged. Instead of defining a language that would have nearly all the components of a general purpose language and,

[†]Supported by Warsaw University of Technology statutory grant 504/036/7 and Program in Control, Information Technology and Automation PATIA.

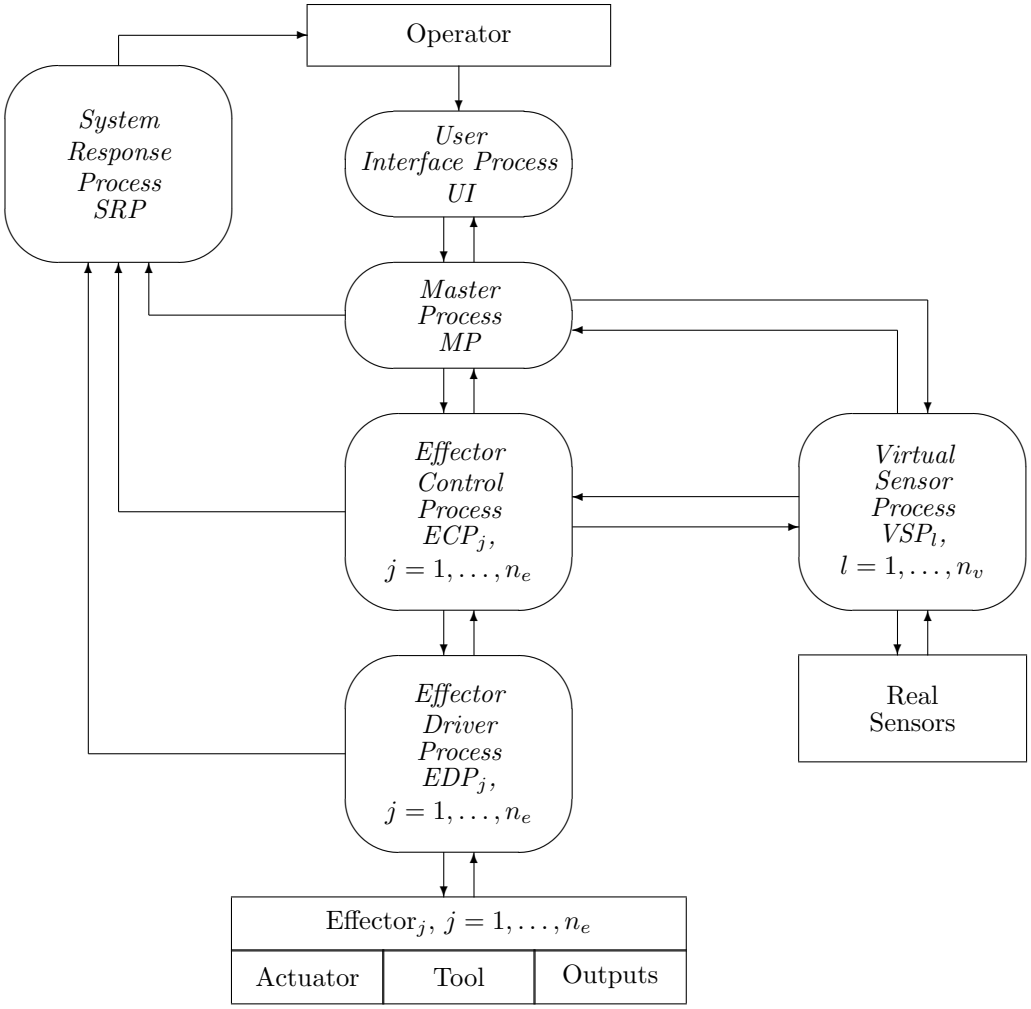

Fig. 1 Structure of a MRROC++ controller (n_e – number of effectors, n_v – number of virtual sensors)

moreover, a few additions taking into account the particular needs of robot system control, it became evident that it is much more convenient to use a universal language and to code the robot specifics as a library of software modules. This idea was followed in: `RCCL` [7], `ARCL` [6], `RCI` [10], `KALI` [8, 9, 2], `RORC` [16, 15], `MRROC` [16, 17]; `PASRO` [4, 5]. Usually procedural programming paradigm has been followed, but currently this changes to object–oriented approach [18]. Moreover, by theoretical investigations, it has been shown that the number of modules in the library can be limited to a very reasonable quantity. The same research pointed out that regardless of the number and type of both robots and sensors the general purpose language has to have a single, but a rather complex, or two much more compact robot instructions as its extensions to become a robot programming language capable of controlling any robotic system and executing any task [20, 21].

2 STRUCTURE OF A MRROC++ CONTROLLER

This paper describes the general structure of an object-oriented version of the Multi-Robot Research-Oriented Controller: `MRROC++`. Theoretical reason for selecting this structure is presented in [16, 20, 21]. This structure is divided into two parts. One is responsible for the execution of the user's task and is robot independent, and the other controls specific robots and is task independent. The paper concentrates on the latter part of this system, namely on the *Effector Driver Process*. Both the software and the hardware component of this part of the system are described on an example of a controller of a prototype robot [3].

As it has already been mentioned, `MRROC++` is a library of software modules (i.e. classes, objects, processes and procedures) that can be used to construct any multi-robot system controller. This set of ready made modules can be extended by the user by coding an extra module in C++. The free-

dom of coding is, however, restricted by the general structure of the system. New modules have to conform to this general structure. Even if a single-robot controller is designed it is assumed that it can work in a multi-robot environment, so its controller really has the capability of controlling several robots. The same applies to sensors. Regardless of the fact, wether they are necessary for the execution of the user's task, the potential for their utilisation always exists in the system.

The `MRROC++` system has a hierarchical structure (fig. 1). It runs on PC computers (Pentium or 486 processor based are preferred) connected by an Ethernet network. This network is supervised by a real-time operating system `QNX-4` [19]. A single process coordinating the operation of the whole sytem is called *Master Process MP*. Each effector (either a robot or a cooperating device) has two processes controlling it: *Effector Control Process ECP* and *Effector Driver Process EDP*. The former is responsible for the execution of the user's task dedicated to this effector, and the latter for direct control of this effector. *EDP* is supervised by *ECP*. In this way the user's task and the effector specific control have been separated and are independent of each other.

Data obtained from real (i.e. hardware) sensors usually cannot be used directly in robot motion control. For instance, to control the arm motion, only the location of the centre of gravity of an object to be grasped would be necessary. In the case of such a complex sensor as a camera a bit-map has to be processed to obtain the above mentioned location. In some other cases a simple sensor in its own right would not suffice to control the motion (e.g. a single strain gauge), but several such sensors deliver meaningful data. The process of extracting meaningful information for the purpose of motion control is named data aggregation and is performed by a virtual sensor. Data aggregation is done by *Virtual Sensor Processes VSP*.

Moreover the system contains two processes dedicated to the interaction with the operator. *User Interface Process UI* handles operator commands. *System Response Process SRP* displays all the system status and error messages on the screen of the monitor. Both processes perform in a windows environment, so operator commands such as: initiation of execution of the user's program, its termination or pausing and resuming are done by clicking on certain icons.

The user's program (task) is coded by writing some distinct portions of *MP* and *ECP*. There are three kinds of tasks that multi-robot systems deal with, namely:

- robots performing independently,
- loosely cooperating robots (e.g. one robot handing an object to the other one),
- tightly cooperating robots (e.g. common transfer of a rigid object over a specified trajectory).

The first kind requires of the *MP* only the initiation and termination of the task. The second requires additionally the synchronisation of the *ECP*s, from time to time. In the last case the *MP* must generate the trajectory for all the robots. In this case the *ECP*s only transfer the *MP* commands to adequate *EDP*s.

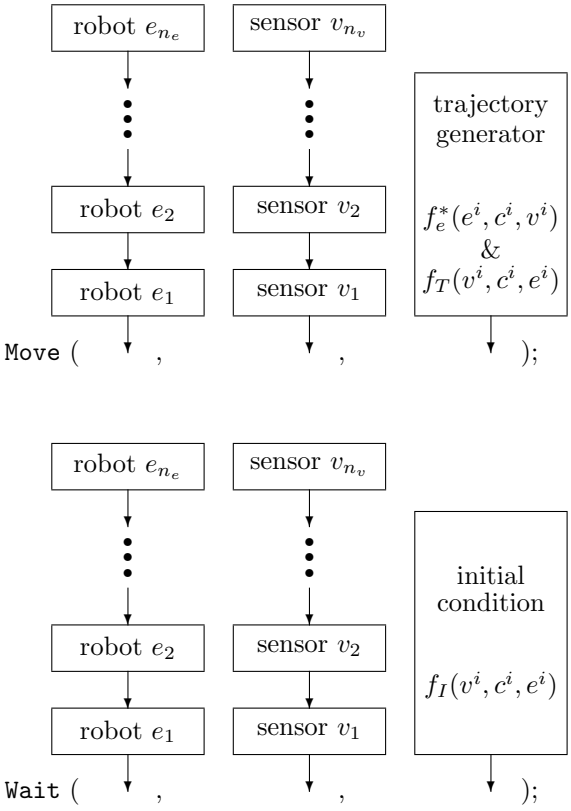

Fig. 2 MRROC++ motion instructions

The user's parts of both the *MP* and the *ECP*s are composed of the `Move` and `Wait` instructions. On the *MP* level these instructions (fig. 2) take as their arguments lists of robots and virtual sensors. On the *ECP* level, for each *ECP* only a single robot exists, so this robot and a list of sensors are the arguments of these instructions. Moreover, the `Move` instruction has an object named trajectory generator as the third argument, and the `Wait` instruction has an object named the condition as the last argument. The generator is responsible, on the *MP* level, for the generation of trajectories of the end-effectors of all the robots on the list forming the argument of the `Move` instruction. On the *ECP* level the generator creates a trajectory for a single robot. The condition, being the argument of the `Wait` instruction, if true, terminates the waiting, and if not causes the system to pause. For each `Move` and `Wait` instruction the user writes in `C++` his or her own generator and condition objects. In this way, usually only very small portions of *MP* and *ECP*s have to be rewritten when the task changes. The modifications are cumulated in the separate code of specific generators and conditions. Errors are dealt with within

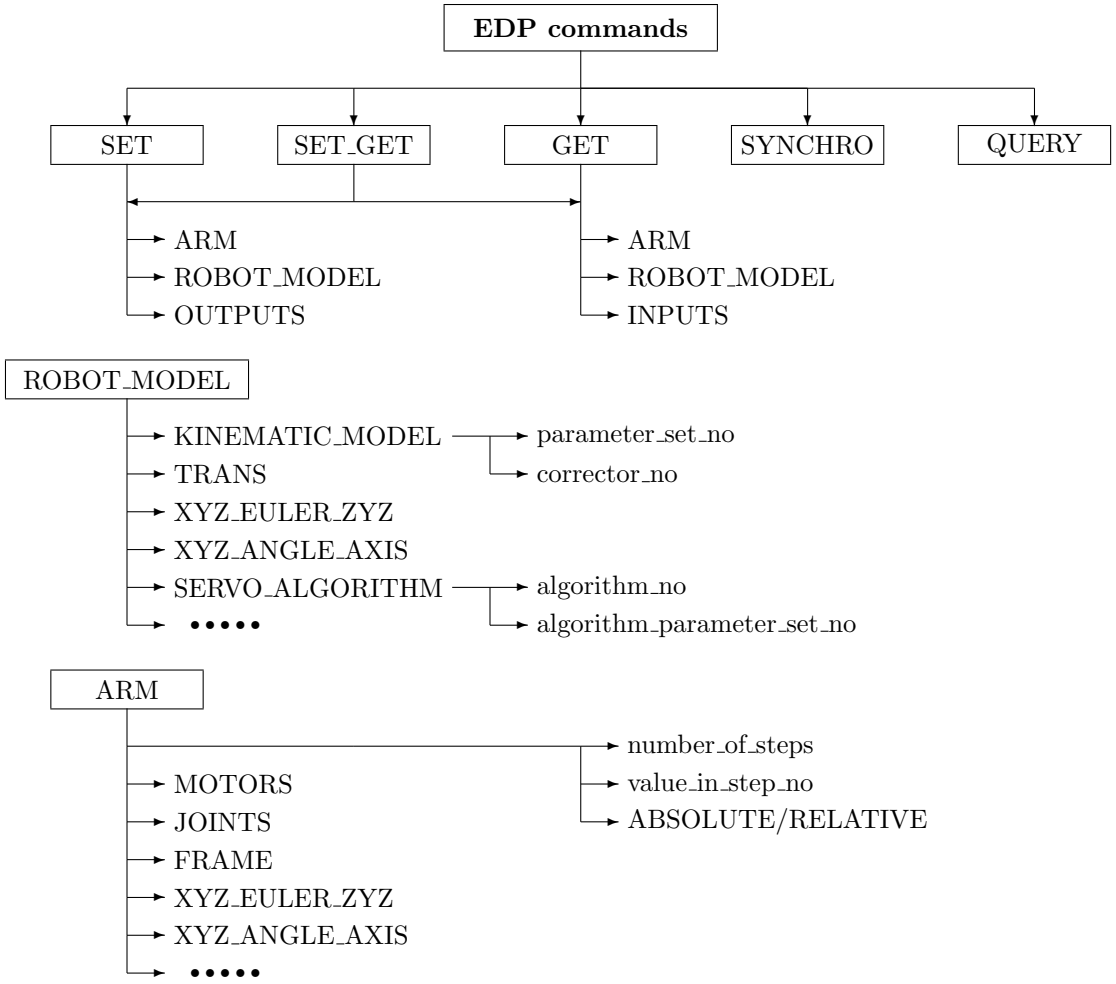

Fig. 3 *Effector Driver Process* commands

the whole system by exception handling, so the user needs not deliver the program code responsible for that. Neither need he or she worry about the code dealing with the inter-process communication.

3 EFFECTOR DRIVER PROCESS

EDP is commanded by its *ECP*. Its code is not modified by the user. It varies only when the effector hardware changes. If the type of the robot is changed, a new *EDP* will have to be supplied, but the user would appreciate, if the remaining part (i.e. the task dependent part) would not be altered. To fulfil this requirement a standard *EDP* communication protocol has been devised. *EDP* is treated as a server interpreting commands issued by all of the remaining parts of the system. Usually only the *UI* and an adequate *ECP* do that. Being a server the *EDP* waits for any client to issue a command. Once the command is delivered, it is interpreted and executed. The list of all commands is presented in fig. 3. If a reply from *EDP* is required another command has to be issued by the client (i.e. `QUERY`). There are two main commands: `SET` and `GET`. The former influences the state of the *EDP* and so the robot, and the latter causes the *EDP* to read its current status. Sometimes, the user needs to exert simultaneous influence on the robot and to read its current state, so a `SET_GET` command has been defined, which causes simultaneous execution of a `SET` and `GET` command. As the majority of robots has incremental position measuring devices, it is required that prior to task execution the robot defines its current position in the work-space. This is usually done only once and by moving the arm to a known base location. This is caused by the `SYNCHRO` command.

The `SET` command can: set the arm position, i.e. cause the robot to move to the desired position, redefine the tool affixed to the arm, change the set of parameters or the local corrector of the kinematic model, switch the servo-control algorithm of any or all of the arm motors, alter the parameters of the servo algorithm, or set the binary outputs of the robot controller. The `GET` command can read:

the current position of the arm, the currently used tool, and the kinematic model and corrector and servo algorithm parameters, or the binary inputs to the robot controller. Switching of kinematic model parameters and correctors should improve local precision of arm motions. Modification of servo algorithms or their parameters can improve tracking ability. This switch can be performed when significant load modification is anticipated.

Both the tool and the arm positions can be defined in terms of homogeneous transforms, Cartesian coordinates with orientation specified either as Euler angles or in angle and axis convention. In the case of the arm position, moreover, it can be specified in terms of joint angles or motor shaft angular increments. The arm position argument in the command can be regarded as an absolute or relative value. Each motion command `SET ARM` is treated as a macro-step. An extra argument specifies into how many interpolation steps it should be divided. Because the incremental position measurement is delivered simultaneously with commanding the new PWM value for the motors, to obtain a continuous motion without stopping, the reading has to be delivered to the upper control layers a few steps before the interpolated motion terminates. The user has control over that by specifying in which step number the reading is required. If this value is one more than the number of interpolation steps, the reading is delivered after the motion stops. For uninterrupted trajectory segment transition it suffices if it is one less than the number of interpolation steps.

The *EDP* is really implemented as two processes: *EDP_MASTER*, which is responsible for client command interpretation, and *SERVO_GROUP* which executes the servo algorithm for all of the arm motors. Currently all the servos are grouped together, but they can be distributed over several processes, if required. The *SERVO_GROUP* process issues its own commands to the robot hardware, as required by the *EDP_MASTER* process. If *EDP_MASTER* does not deliver any commands the *SERVO_GROUP* enters a passive loop, where it sustains a zero motion increment command, causing the arm to remain immobile.

4 CONTROLLER HARDWARE

Each DC electric motor, actuating a degree of freedom of the arm, is controlled by a separate axis controller. The axis controller consists of: a control microcomputer based on the one-chip MCS-51 family 80552 microcomputer, a position measurement circuit, a PID regulator and an interface circuit (fig. 4). Axis controllers are connected to a PC type master computer through an interface in such a way that interfacing circuit registers are mapped into the input-output address-space of the master computer. The master computer that runs the *EDP*. The microcomputer communicates with it and interprets its commands.

The axis controller has two modes of operation. In the first one, the servo algorithm is implemented in the master computer by the *SERVO_GROUP* process, so the microcomputer generates the PWM signal and measures the current position by using a specialised 32-bit reversible counter. In the second mode, besides performing the above functions it also serves as a servo-regulator. For this purpose it uses National Semiconductors LM629 PID controller. In this case the *EDP* running on the master computer does not have to execute the servo algorithm, so it saves the computation power for other tasks. Obviously, in this case the algorithm is limited to PID control only, although its parameters can be modified. When the LM629 is used the position measuring counter is not active. In both modes of operation the microcomputer continuously monitors the state of limit switches and the value of the DC motor current. Activation of a limit switch or transition over the threshold current in the motor causes immediate motor stop and an error message being sent to the master computer, i.e. to *EDP*.

5 CONCLUSIONS

The system is currently undergoing intensive testing. Several specialised controllers have been designed using the `MRROC++` language/library and methodology. To check, if this methodology can be used to design an industrial controller the trajectory teach-in and playback capability have been introduced into the system.

Another specialised controller has been designed specially for the purpose of kinematic model calibration. In this case, first, special calibration trajectories are recorded, and then they can be played back as many times as necessary. In each intermediate pose the arm stops. Position expressed in motor shaft increments and in Cartesian coordinates and Euler angles is recorded. A program running on another computer coupled to two electronic theodolites creates a file with external coordinates of the end effector in each of the intermediate trajectory poses. As the calibration trajectory is stored in a file it can be reproduced as many times as necessary, so the calibration process can be repeated under different working conditions (e.g. load, just after initiation of robot operation, after a long period of operation, in different temperatures).

The capability of tight and loose cooperation have been tested by simulating the second robot. When work on the system finishes an IRp-6 robot will be the partner of the prototype robot for which the `MRROC++` controller has been made. It is envisaged that the prototype robot, because of its unique

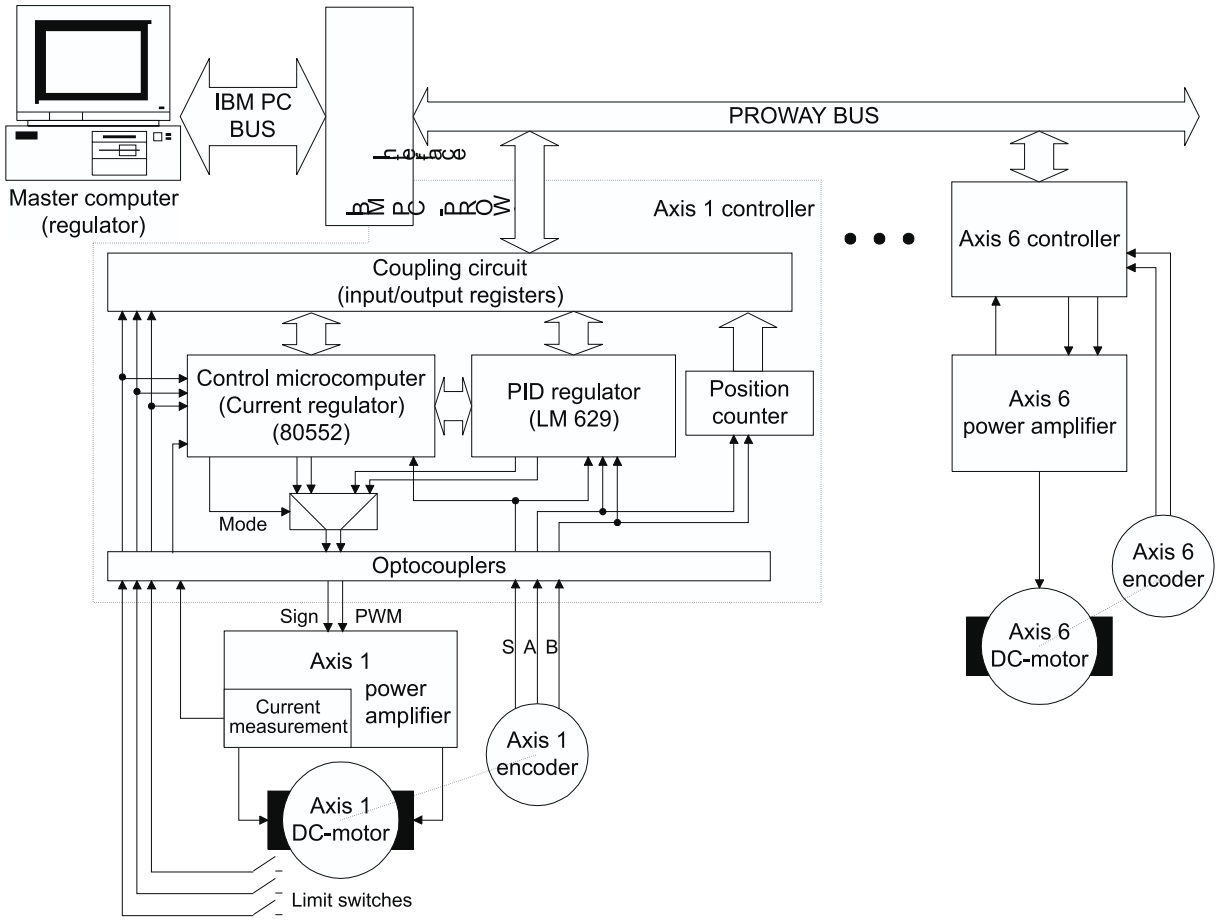

Fig. 4 Axis controller

property: high stiffness, can be used for milling. A controller dedicated to this task is also undergoing tests now.

REFERENCES

[1] Ambler A. P., Corner D. F.: *RAPT1 User's Manual*. Department of Artificial Intelligence, University of Edinburgh, 1984.

[2] Backes P., Hayati S., Hayward V., Tso K.: *The KALI Multi-Arm Robot Programming and Control Environment*. Proc. NASA Conf. on Space Telerobotics, 1989.

[3] Bidziński J., Mianowski K., Nazarczuk K., Słomkowski T.: *A manipulator with an arm of serial parallel structure*. Archives of Mechanical Engineering, Vol.39, No.1-2, 1992, pp.65-78.

[4] Blume C., Jakob W.: *PASRO: Pascal for Robots*. Springer-Verlag, Berlin 1985.

[5] Blume C., Jakob W.: *Programming Languages for Industrial Robots*. Springer-Verlag, 1986.

[6] Corke P., Kirkham R.: *The ARCL Robot Programming System*. Proc. Int. Conf. Robots for Competitive Industries, Brisbane, Australia, 14-16 July 1993. pp.484-493.

[7] Hayward V., Paul R. P.: *Robot Manipulator Control Under Unix RCCL: A Robot Control C Library*. Int. J. Robotics Research, Vol.5, No.4, Winter 1986. pp.94-111.

[8] Hayward V., Hayati S.: *KALI: An Environment for the Programming and Control of Cooperative Manipulators*. Proc. American Control Conf., 1988. pp.473-478.

[9] Hayward V., Daneshmend L., Hayati S.: *An Overview of KALI: A System to Program and Control Cooperative Manipulators*. In: *Advanced Robotics*. Ed. Waldron K., Springer-Verlag, 1989.

[10] Lloyd J., Parker M., McClain R.: *Extending the RCCL Programming Environment to Multiple Robots & Processors*. Proc. of the IEEE Int. Conf. Robotics & Automation, 1988. pp.465-469.

[11] Mujtaba S., Goldman R.: *AL Users' Manual*. Stanford Art. Int. Lab., 1979.

[12] Paul R.: *WAVE: A Model Based Language for Manipulator Control*. The Industrial Robot, March 1977, pp.10–17.

[13] Taylor R. H., Summers P. D., Meyer J. M.: *AML: A Manufacturing Language*. Int. Journal of Robotics Research, Vol. 1, No. 3, 1982.

[14] Zieliński C.: *TORBOL: An Object Level Robot Programming Language*. Mechatronics, Vol.1, No.4, Pergamon Press, 1991. pp.469-485.

[15] Zieliński C.: *Flexible Controller for Robots Equipped with Sensors*. 9th Symp. Theory and Practice of Robots & Manipulators, Ro.Man.Sy'92, 1-4 Sept. 1992, Udine, Italy, Lect. Notes: Control & Information Sciences 187, Springer-Verlag, 1993. pp.205-214.

[16] Zieliński C.: *Robot Programming Methods*. Publishing House of Warsaw University of Technology, 1995.

[17] Zieliński C.: *Control of a Multi-Robot System*, 2nd Int. Symp. Methods & Models in Automation & Robotics MMAR'95, 30 Aug.–2 Sept. 1995, Międzyzdroje, Poland. pp.603-608.

[18] Zieliński C.: *Object-Oriented Robot Programming*, Robotica, Vol.15, 1997. pp.41–48.

[19] *QNX System Architecture*. Quantum Software, 1992.

[20] Zieliński C.: *Object–Oriented Programming of Multi–Robot Systems Utilising Sensory Information*. 3rd ECPD Int. Conf. Advanced Robotics, Intelligent Automation and Active Systems, 15–17 September 1997, Bremen, Germany, pp.176–181.

[21] Zieliński C.: *Object–Oriented Programming of Multi–Robot Systems*. 4th Int. Symp. Methods and Models in Automation and Robotics MMAR'96, 26–29 August 1997, Międzyzdroje, Poland, pp.1121–1126.