Versatile control system for a very fast robot [*]

URSZULA KRÊGLEWSKA, KRZYSZTOF SACHA

Warsaw University of Technology, Institute of Control and Computation Engineering, Warsaw, Poland, ula@ia.pw.edu.pl, k.sacha@ia.pw.edu.pl

Abstract. The paper describes the hardware and software architecture of a multicomputer control system of a very fast robot. The control system was designed for research and educational purposes with two goals in mind. First, it had to be flexible and modifiable with respect to control algorithms applied to particular axes. Second, the interfacing of the system to additional devices, such as environmental sensors and other robots, should be possible with only limited effort. The various constraints and considerations imposed by these goals are examined. In addition, the technical problems related to the use of direct drive motors are described, and the impact of the requirement for very fast operation on the system architecture is discussed.

Key Words. Robot control, hard real-time control, multiprocessor control system.

1. INTRODUCTION

The paper describes the hardware and software architecture of a multicomputer control system designed for an experimental very fast robot. The robot column has been designed in the Institute of Aeronautics and Applied Mechanics. This is a six degree-of-freedom robot with three main axes driven by powerful high torque direct drive motors, and three other axes driven by much smaller conventional alternate current motors.

The main axes drive systems comprise a direct drive motor and motor driver (manufactured by Yokogawa), with the motor being coupled to the arm without any gearbox. The range of the arm movement is unlimited in that the arms can revolve many times around their axes. The speed of the direct drive motors ranges up to 2 revolutions per second, and the accuracy of the motor positioning — 0.0007° (500 000 distinct positions per revolution).

The robot is designed mostly for research and educational purposes. The users of the device are likely to play with different types of control algorithms for particular axes. This imposes an additional requirement on the control system of the robot — it should be very flexible and a modification of the control algorithm should be possible with only limited effort.

2. LOGICAL ARCHITECTURE

The general software structure of the robot controller is shown in fig. a [6]. The upper layer task, called MASTER, coordinates the robot movement, i.e. it computes the *kinematic model* of the robot and decomposes an elementary step of the robot column into the individual steps of particular robot axes. The lower layer tasks, called SERVO, implement control algorithms, such as e.g. PID or MRAC, of particular axes. The axis control can be based on setting the

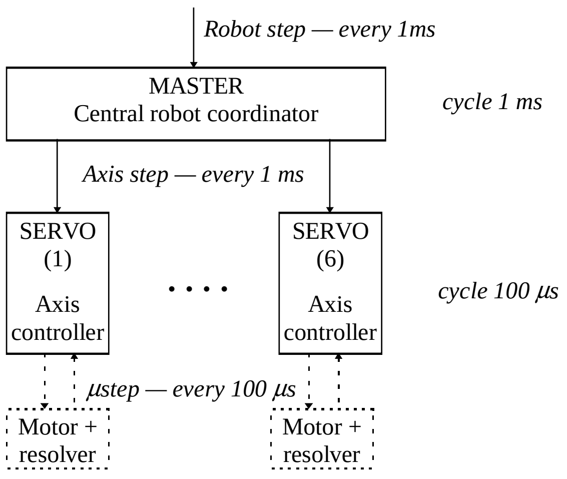

Fig. A. Control software architecture

[*]) The research was supported in part by Committee for Scientific Research under the grant no 8 T11A 028 09

desired velocity or torque of the motor. Current arm position is fed back to the controller as counts from the motor resolver. Other status data can also be read from the motor driver.

High frequency of the computation is required to take advantage of the possible speed and accuracy of the robot motors: MASTER task has to be repeated cyclically once per a millisecond, and the calculation of each SERVO control tasks has to be repeated ten times per one millisecond. These requirements create the need for a multiple computer architecture of the robot control system.

Other tasks can also be performed by the robot control system, in particular, the tasks for graphical man-machine interface, for communication with advanced environmental sensors, such as video cameras, and for supervisory control of a multirobot workcell. These additional tasks can be executed either in the background of the basic robot control activity in a multitasking (time sharing) environment, or can be shifted to additional computers in a distributed environment. The latter solution is more complicated and slightly more expensive, but can offer better performance and more modifiable system and application structure.

3. HARDWARE ARCHITECTURE

The requirements on the robot control system, which can be derived from the general objectives of the project stated in Sections 1 and 2, are for:

- high reliability, as the system is to be used in an educational environment,
- high flexibility and modifiability,
- high performance, as a very short repetition cycle of the axis controllers is needed.

The first of those requirements (high reliability) eliminates the use of popular PC-type computers. Economical limitations and the requirement for the ease of modification eliminate the use of DSP and transputer systems. What remains is an architecture composed of universal microprocessors of a modular industrial computer kit.

The required performance of the control system cannot be met by a single microprocessor. Therefore, a three processor architecture has been developed, with a single processor running MASTER task and two other processors running SERVO tasks of three robot axes each.

The computational characteristics and requirements of both types of tasks are different, as MASTER task (which computes the kinematic model of the robot) uses mostly trigonometric functions, and SERVO task (which implements a control algorithm) uses mostly simple arithmetic operations. This suggests the need for two different types of processors used for running different tasks.

The computer hardware selected for the development of the robot control system is based on an industrial VME-based computer kit from PEP Modular Computers. The hardware is able to work in a multiprocessor configuration composed of Intel 486 and Motorola 68060 processor boards. Basic characteristics of these processors, published in commercial papers, were not sufficient for comparison. Hence, an experiment was performed, in which the particular arithmetic operations were executed in long loops. The number of loop cycles was counted and the duration of the loop was measured. The results of the measurements are shown in table a. The experiment proved that 486 could better perform in MASTER layer, while 68060 in SERVO layer.

Table A. Characteristics of the processors

Type of operation	486/66Mhz [Mop/s]	68060/50Mhz [Mops/s]
Fixed point	3.1	3.6
Floating point	1.6	2.7
Trigonometric	0.3	0.2

The selected architecture of the robot control system consists of a single Intel 486 processor board and two Motorola 68060 processors boards connected to VME bus (fig. b) and mounted within a single rack, together with nearly fifteen interface circuit boards. The processors can communicate with each other through common memory areas accessible via VME bus and can exchange interrupt signals. Both 68060 processors act as axis controllers and execute SERVO control tasks, while 486 performs the supervisory MASTER task. Moreover, the supervisory processor can cooperate through a network with external computers, thus creating a distributed control system with enormous power and flexibility.

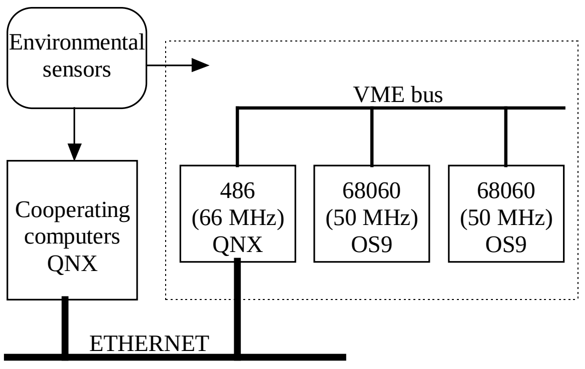

Fig. B. Hardware architecture of the robot control system

The interface circuit boards that link particular processors with their axes are referenced by the processors. The needs of the control algorithm and the type of the interface boards are such that up to ten words can be transferred within a single SERVO task cycle. Since the transfer of a data word from an interfacing board takes about 500ns, the total load produced by the controllers of 6 axes can reach 30µs within a 100µs cycle. If the interfacing boards were placed directly on VME bus, this load could consume 30% of the total bus bandwidth. Taking into account additional load introduced by the interprocessor communication, one can note that the bus could become a bottleneck in a single bus architecture, which could result in violation of the required repetition cycle.

To prevent traffic jams on the communication system within the control computer, a dual bus architecture has been selected in which the interface boards are coupled to local buses of particular axes processors (fig. c). VME bus is used as the main system bus for inter-processor communication only. Relatively low actual load on the main bus preserves the potential for future system expansion, as additional robot-related sensors, not assigned to particular axes, can be interfaced to the supervisory processor through VME bus. This is the only possibility for such expansion, as no local buses for 486 board are offered.

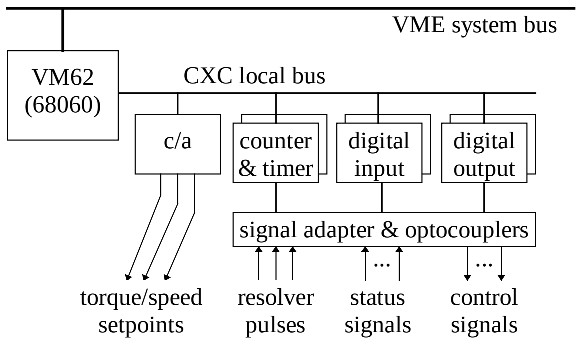

Fig. C. Hardware structure of an axis controller

The interface to a motor driver consists of an analog torque or speed setpoint signal, quadrature pulse signals from relative resolver and a number of bistable status and configuration signals. An external synchronization sensor supplements each axis hardware. Optocouplers, shown in fig. c, are indispensable, as the total rush current of the high-torque direct drives can exceed 100A and can reproduce a dangerous current flow through galvanic connections throughout the interfacing circuits.

4. SOFTWARE ARCHITECTURE

The operation of Intel 486 is controlled by QNX, a multitasking, distributed real-time operating system from QNX Systems Software [3,5]. The operation of 68060 is controlled by OS-9, a multitasking, real-time operating system from MicroWare [2]. The systems differ from each other in many aspects and cannot cooperate directly using their standard tools for inter-task communication. Basic sets of these tools and the performance characteristics of both systems, partially measured [4] and partially found in the literature [1], are listed in table b. The comparison of the data shows, that the speed of the system operation is in both cases similar.

Table B. Characteristics of the operating systems

QNX			OS-9	
Operation	Duration		Operation	Duration
task switch	6 µs		task switch	5 µs
signal	45 µs		signal	20 µs
proxy	25 µs		event	15 µs
semaphore	15 µs		semaphore	14 µs
message	20 µs		interrupt	18 µs

The performance requirements formulated in Section 2 impose hard real-time constraints on the robot control operation, and particularly on the operation of the axes controllers implemented by 68060/OS-9 computers. These requirements exceed, in fact, the real possibilities of a multitasking operating system (but not the possibilities of the hardware). In case of OS/9, the standard configuration is offered with the real time clock resolution equal to 10ms. Special configuration with the clock resolution of 100µs can be delivered on request. However, better resolution results in an increased overhead, which decreases the overall system throughput.

The resolution of the real time clock is not the only problem encountered in designing the software. Axis controllers (SERVO tasks in fig. a) are basically independent of each other — the coordination between the axes is ensured by the supervisory MASTER task. Hence, the most flexible and modular structure of the 68060 software could consist of 3 separate tasks, each of which could control a separate robot axis. The execution of the tasks could be triggered by a standard OS/9 tool for cyclic operation, i.e. a cyclically generated signal. However, each SERVO task must be executed once per 100µs. A quick look to the table b shows that the speed of signal generation and delivery is too slow for the required 100µs repetition cycle.

To solve the problem, a variant of a two-level, foreground-background software architecture has been developed. Control algorithms (SERVOs) of the three axes are executed sequentially within a single task, which runs at a high priority level. The execution of the control task is triggered by a service function of an interrupt generated cyclically by a timer device. Such a design eliminates the need for a high resolution operating system configuration. The communication of the axes control task with the robot coordinator (MASTER, executed by another processor) is implemented by a separate communication task, which runs at a low priority level (fig. d). The control task and the communication task can exchange data and synchronize to each other through a common data area. The synchronization uses simple flag-based handshaking, and is relatively simple, as only the control task can interrupt the communication task, but not vice versa.

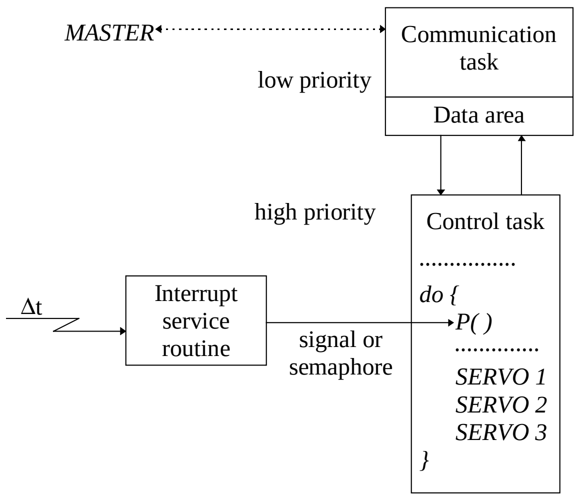

Fig. D. Foreground-background architecture

Nice feature of the architecture shown in fig. d is simplicity. Vital functions of the control algorithm are executed at the task level, the pattern of inter-task communication is simple, and the use of interrupts is reduced to the minimum. This makes software development and software debugging relatively simple and efficient.

Unfortunately, this architecture could be used only in the initial phase of our work, as it did not meet the performance requirements. The real measurements showed, that the figures in table b described the mean values only, while the worst case delays were much longer. In particular, the worst case signal delivery exceeded 100µs, thus eliminating any possibility of the use of system tools within the axes control loop. Therefore, the architecture in fig. d had to be modified in such a way that the control loop was shifted to the body of the interrupt service function. The modification, shown in fig. e, had no significant influence on the structure of the communication task.

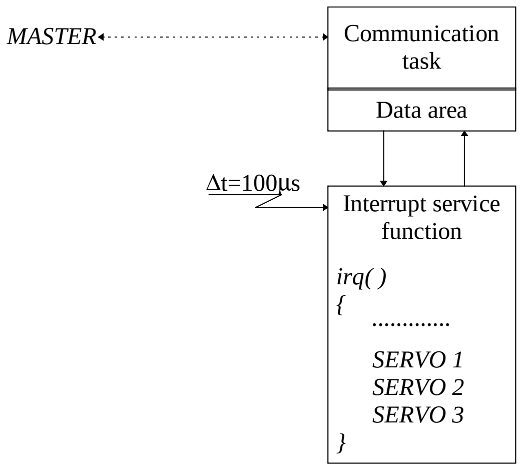

Fig. E. Target architecture of the axes controller

Real-time constraints imposed on the operation of the robot coordinator implemented by 486/QNX computer are less critical, as MASTER task must be executed only once per 1ms. Such a repetition rate can be matched by QNX, which can operate with the time resolution equal to 0.5ms. Moreover, a violation of the required repetition cycle is also less critical, as it can slow down the robot operation but cannot cause any dangerous robot movement. Hence, software architecture of the robot coordinator can be based on standard multitasking features of QNX, with standard system tools being used for intertask communication.

The cooperating computers in fig. b are PC-clones operating under QNX. The computers work in real-time, but the constraints are, in general, less severe than the constraints imposed on the robot coordinator (486/QNX). The communication of the robot coordinator and the cooperating computers is implemented by means of standard QNX tools for intertask communication. Since QNX is a distributed operating system with a full possibility for task communication and resource sharing within a local area network, no special considerations with respect to this aspect of system architecture are needed.

A separate problem is the communication between the robot coordinator (486/QNX) and axis controllers (68060/OS-9). The needs of the application are for two types of tools:

- messages, for transferring commands from the robot coordinator to axis controllers, and a status data in the reverse direction,
- signals, conveyed in both directions, for exception handling.

Neither of those tools are available. Moreover, system software does not support inter-processor communication at all. The computer hardware offers a possibility of physical access to common memory areas with no means for synchronization, and a possibility of mutual interrupting. No tools for higher level communication are implemented. The problem must be solved entirely within the application. Because it is crucial for the system modifiability, it will be considered in detail in a separate section.

5. MODIFIABILITY AND COMMUNICATION

The requirements specification of the very fast robot control system includes a requirement for high flexibility and modifiability. The intention of the project is that neither hardware nor software architecture is constant within the whole system lifecycle. At the hardware level this means that the number of processors can change, and an additional 68060 can be attached to give more computational power for the axis controllers. At the software level this means that particular axis controllers (SERVOs in fig. a) can be shifted and assigned to different processors within the control system — in other words, the code modules which compute axes control algorithms can be shifted from one processor to another one. The contents of those code modules can also be changed.

Any change in the assignment of axis controllers to processors affects the communication between the robot coordinator and the axes control processors. If the low level communication mechanisms (common memory areas and inter-processor interrupts) were used directly, the impact of such changes on the structure of the robot coordinator as well as axes control software could be destructive, in that any change could require a huge number of small modifications to the software code.

The problem of changes identified above can be solved by implementing high level tools for message and signal passing, with the potential for redirection, following any reconfiguration of the hardware and software structure. Such a design makes the application software of the robot coordinator (486/QNX) and cooperating computers (fig. b) independent of the actual structure of the axes control level.

The implementation is based on a concept of a *port*, which can be interpreted as a uni-directional communication channel between a sender task and a receiver task. Ports are identified by numbers, and described in a port table stored in a common memory area (fig. f). A port description consists of a port number, the receiver task identifier (process identifier — *pid*) and a number of the processor, which executes the receiver task. The assignment of ports to receivers is constant, e.g.:
port 1 — status and signals to MASTER,
port 2 — commands and signals to SERVO 1, ...

Port nr	Processor nr	Receiver id	
1	1	pid_M	→ MASTER
2	2	pid_{S1}	→ SERVO 1
...	...	...	

Fig. F. Port table

The port table is filled in during system start-up: The processor numbers are written manually as configuration data, while the receiver task identifiers (*pid*s) are written automatically by the code modules of MASTER and SERVOs. This way any command or signal to a particular application module (MASTER or SERVO) can always be sent to the same port, regardless of the actual placement of the module.

Two basic primitives of port-based communication have been implemented:
- *send*, for message transfer,
- *signal*, for exception signalling.

The implementation is divided between library functions and the communication tasks. Basic structures used by the implementation of the function *send*, which sends a message from MASTER to SERVO are shown in (fig. g). A call to the function *send* is processed within the following sequence:
1. The function looks to the port table and finds the receiver processor nr and the receiver id assigned to the port *p*. Next, it places a message (port nr + receiver pid + data *c*) in the receiver processor queue and sends an interrupt request to the receiver processor.
2. The interrupt service function executed by the receiver processor posts the semaphore *s* and resumes the communication task.
3. The communication task reads the message from the queue and stores the data *c* in the data area in a slot assigned to the received port number.

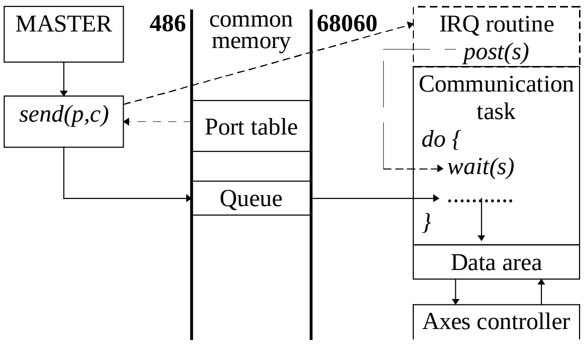

Fig. G. Implementation of the function *send*

The implementation of the function *signal* works similarly. The main difference lies in that no data is passed and the communication task does not refer to the data area, but just sends a signal to the receiver task (by calling the standard *kill* function of QNX or OS-9). The function *signal* can be called by an application task or by an interrupt service function executed by any processor of the system.

The function *send* is not used directly by the application programs. Instead, it is used as a building block for implementing rendezvous, semantically similar to the standard QNX message operation. Rendezvous is implemented by a function *exchange* and consists of a pair of *send* functions which convey data in the oposite directions. Rendezvous can be called by 486 tasks only. It is used by MASTER task as a confirmed communication service for sending a command to a selected SERVO task and receiving back a status data. Signals can be raised by 486 as well as 68060 programs. They are used as unconfirmed services for signaling in both directions exceptional situations which emerged during the program execution.

The described implementation is completely safe in that neither a message nor a signal can be lost. The worst case arises if a second interrupt was requested before the previous one had been accepted. In such a case only one interrupt is effectively generated, while the second request is lost. The interrupt service function within the receiver processor anticipates such a situation and always posts the semaphore as many times, as is the number of messages waiting in the queue. Thus, if two messages were written to the queue at nearly the same time, both of them are processed regardless of the number of interrupts received. The communication between the interrupt service function and the communication task uses a semaphore which counts the number of messages or signals stored in the queue, so that neither of them can be lost.

6. DEVELOPMENT ENVIRONMENT

The development of software requires appropriate hardware and software environments, comprising disk memories, terminals, editors, compilers, debuggers, etc. All these facilities exist in our research system. The development system configuration differs from the one shown in fig. b in that all processors have network interfaces and can be accessed via standard Tcp/Ip services (ftp, telnet). Tcp/Ip services (nfs) are also used by particular 68060 processors to access a single disk memory station with a set of software tools.

Neither of the facilities mentioned above are needed in the target configuration. Moreover, the development configuration cannot be used in the target control system, as network software of OS-9 does not perform well in hard real-time. Network interrupts and system-state processes introduce a significant overhead and extend the worst case interrupt reaction time to above 100µs, and the signal delivery time to above 1ms. Therefore, the entire network software has to be unlinked from the target OS-9 system configuration.

7. CONCLUSIONS

The control of a high resolution fast robot is a challenging task which nearly reaches the limits of the current microcomputer technology. Meeting the hard real time constraints requires hardware architecture with multiple processing units and multiple buses for internal data transfer within the control system. Software architecture at the axes control level cannot benefit the advantages of multitasking operating system, but must be tailored to meet severe real time limitations.

The system described in our paper is currently under development. The computer hardware has been assembled and debugged, basic software structures and control algorithms have been fixed and prototyped. A complete working robot system is expected at the end of this year.

8. REFERENCES

1. Zieliński C.: Object-oriented robot programming, Robotica, vol. 15, 1997, pp. 41-48
2. QNX System Architecture, Quantum Software Systems, Kanata, 1992.
3. OS-9 Technical Manual, Microware Systems Corporation, Des Molnes, 1993.
4. Sacha K.: Measuring the Real-Time Operating System Performance, Proc. 7th Euromicro Workshop on Real-Time Systems, 1995, pp. 34-40
5. Sacha K.: QNX − System operacyjny, X-Serwis, Warszawa, 1995 (in Polish)
6. Dibble P.: OS-9 Insights, An Advanced Programmers Guide to OS-9, Microware, 1994