Robotica (2004) volume 22, pp. 661–679. © 2004 Cambridge University Press
DOI: 10.1017/S0263574704000360 Printed in the United Kingdom

Reconfigurable distributed real-time processing for multi-robot control: Design, implementation and experimentation

J. Z. Pan* and R. V. Patel†

(Received in Final Form: February 11, 2004)

SUMMARY

Sophisticated robotic applications require systems to be reconfigurable at the system level. Aiming at this requirement, this paper presents the design and implementation of a software architecture for a reconfigurable real-time multiprocessing system for multi-robot control. The system is partitioned into loosely coupled function units and the data modules manipulated by the function units. Modularized and unified structures of the sub-controllers and controller processes are designed and constructed. All the controller processes run autonomously and intra-sub-controller information exchange is realized by shared data modules that serve as a data repository in the sub-controller. The dynamic data-management processes are responsible for data exchange among sub-controllers and across the computer network. Among sub-controllers there is no explicit temporal synchronization and the data dependencies are maintained by using datum-based synchronization. The hardware driver is constructed as a two-layered system to facilitate adaptation to various robotic hardware systems. A series of effective schemes for software fault detection, fault anticipation and fault termination are accomplished to improve run-time safety. The system is implemented cost-effectively on a QNX real-time operating system (RTOS) based system with a complete PC architecture, and experimentally validated successfully on an experimental dual-arm test-bed. The results indicate that the architectural design and implementation are well suited for advanced application tasks.

KEYWORDS: Real-time processing; Multi-robot control; Fault detection; Reconfigurable processing; Home-care system.

1. INTRODUCTION

One of the driving forces behind incorporating robotic systems into a complex application is that the robots can be programmed for various tasks. Such programmability, or reconfigurability at the task level, is made possible by a proper design of the robot control system. Traditionally, when a robot system is delivered, the structure of its control system is fixed. Although in a multi-robot system, robots can be re-arranged to do different cooperative tasks based on the programmability of the individual robot, the control system itself is not designed for reconfiguring the structure in the scope of a multi-robot system as a whole.

A traditional approach for building robot control systems is to develop a controller for a specific type of robotic system. Fiorini et al.[1] described a PC-based motion controller using the Configuration Control algorithm for the RRC manipulator. Central to the controller is an interrupt handler design (serving as a system scheduler) by combining features of the iRMX real-time operating system (RTOS) and one of the programmable counters of the PC. Several controller processes run at different frequencies. The system was previously implemented on a VME-bus/VxWorks RTOS for high-level control and a Multibus-based system for joint-level control[2,3] and was re-implemented on a PC and Multibus-based system. The two sub-systems were connected via a VME-to-Multibus or PC-to-Multibus extender. A similar system structure (a VME-bus and PC-based system with a bus extender) was adopted to develop and implement a control system for a dual-arm redundant robot system.[4]

As for multi-robot control systems, an attractive approach for architecture design is using multi-agent systems with distributed artificial intelligence theory. Fraile et al.[5] presented a distributed planning and control architecture for autonomous multi-manipulator systems using a multi-agent paradigm. The framework was aimed at flexible robotic assembly tasks. Issues of flexibility, scalability, reconfigurability, and fault-tolerance were addressed by deploying a team of distributed and autonomous agents negotiating, collaborating and cooperating to achieve the goals. A designated communication format and protocol was used by the agents to exchange information. All system components were modeled and constructed in a uniform and homogeneous manner. The agent-based control system was implemented in a distributed fashion on four PCs, each of which was connected to one of the low-level manipulator controllers over a serial RS-232 line. All four PCs were further linked through a local Ethernet network. Issues related to real-time performance were not reported.

Being aware that developing robust real-time software can consume the major resources of a project, Roberts et al.[6] conceived a type of software architecture for robotics and automation that addresses modularity and re-usability of the system components. The issues of safety, off-line testing and realistic simulation were addressed. Two-level concurrency of the components was used: coarse grain with operating system processes and fine grain using threads. Remote

* Department of Electrical & Computer Engineering, University of Western Ontario, London, Ontario (Canada) N6A 5B9; currently a Design Engineer at Dr. Robot Inc., Toronto, Ontario (Canada).
† Department of Electrical & Computer Engineering, University of Western Ontario, London, Ontario (Canada) N6A 5B9.

procedure calls and message queues were used for inter-process communication. System safety and reliability were achieved using combined hardware and software features. The architecture was implemented on a VME-bus based system for a large mining machine – a 3500 tons dragline. All the components ran at frequencies ranging from 1 Hz to 10 Hz. Issues concerning system scalability and reconfigurability were not reported. Traub and Schraft[7] presented a real-time framework for distributed control systems that is aimed at increasing software modularity, portability, and reusability. It provides an interface to operating system dependent I/O operations for TCP/IP, serial communication and CAN field-bus, which are based on the client-server structure. The client-server communication connects not only different hardware units but also processes and threads running on the same machine. Applications based on this framework are shielded from any operating system dependence. Design patterns are developed in order to express and document common structures on a micro-architectural level between different applications. This real-time framework was designed with UML and implemented in C++ on Windows NT and Windows CE. The paper also describes the use of the framework as a basis for the control software of a mobile home-care system Care-O-bot.[8]

Recently, increasingly sophisticated applications, such as multiple cooperative robots, macro-micro manipulators, autonomous mobile robot colonies, and flexible manufacturing and assembly systems, are attracting the attention of many researchers and organizations. The nature of these new systems challenges the traditional control system in that the system structure needs to be more flexible and, if reconfigured, the system should be able to maintain a reliable real-time performance. The system should be reconfigurable and scalable in both control hardware and software when a requirement of functionality changes. The new incorporated components may run at a different frequency, share run-time information, and contribute to the enhanced system functionality, while the existing components should stay intact if so required. A more advanced and powerful control system is needed to accommodate such applications.

This paper describes the work on design and implementation of a software architecture for a reconfigurable distributed real-time multi-processing system for multi-robot control. Several essential points such as modularity, reusability and mobility of the system components have been addressed. Much effort has been made to construct a robust and flexible yet cost-effective control system. The system is implemented on a QNX RTOS-based system[9,10] with a PC architecture, and has been successfully validated experimentally on the REDIESTRO dual-arm test-bed. The results indicate that the architectural design and implementation are well suited for advanced application tasks.

This paper consists of seven sections. Section 2 elaborates on the motivations and requirements. Section 3 presents the analysis and modeling of the multi-robot control system. Section 4 describes the system architectural design. Section 5 discusses the key issues of a QNX-based implementation. Section 6 presents the results of the experimental tasks performed by the implemented system on the REDIESTRO dual-arm system. The paper draws conclusions in Section 7.

2. MOTIVATIONS AND DESIGN REQUIREMENTS

This paper deals with a software architecture design of a complex control system for multiple robots in a research environment. In addition to meeting basic functional requirements, the system architecture should ensure that the special requirements, such as scalability and adaptability, schedulability and predictability, safety and reliability, and cost-effectiveness, are realizable.

2.1. Scalability and adaptability

The scalability requirement arises from the need to use the same system structure for different robot applications (e.g. dual arm systems and macro-micro manipulators). The control system should have the capability to be reconfigured at the robot level, and/or at the joint (or sensor and actuator) level. They should also adapt to various numbers of robots (homogeneous and/or heterogeneous) incorporated in the system. When the scale of the system increases, it may come to a situation where running all the system components on the originally designed hardware platform will result in unsatisfactory performance. This is the adaptability requirement that needs the control system to be hosted on different topologies of computer systems, from a single computer to a distributed computer network.

In order to meet the scalability and adaptability requirements, the whole system functionality should be partitioned and modularized. The information exchange scheme among the system components should be efficient yet flexible, so that all the components can work in an autonomous or loosely coupled way. Modularity is also helpful for system development and maintenance.

2.2. Schedulability and predictability

A robot system usually characterizes itself as a hard real-time system, when used in industrial and research environments. The effectiveness of its performance is characterized by the logical correctness of its behavior and the physical timing of the desired behavior as well. In other words, late sensing data or control responses mean a system failure. The consequence may bring the robot into unstable states, which may result in a catastrophe at the cost of the robot system, its environment, and even the human operators.

In order to meet the hard real-time requirement, different activities of the application should be schedulable and system behaviors should be predictable.[11] The system should be logically partitioned not only for accomplishing the overall functional requirement but also for good scheduling. Proper prioritizing of the system sub-modules is essential for supporting correct synchronization.

2.3. Reliability and safety

Besides being a time-critical system, a robot system is also a safety-critical system. As robot systems become more complex, reliability and safety issues become increasingly important in the system design. The system should be robust enough with reliable performance and fail safe operation. It should work well in an environment with reasonable disturbances and gracefully degrade in the presence of a partial system failure. In addition to being deterministic in the time domain, the system should be deterministic in its behavior and the consequences of its behavior. Particularly, the energy transferred from the system to the environment should be bounded. This includes momentum and forces executed by the robots.

In order to meet the reliability and safety requirement, the system should ideally have fault-tolerance and fault-recovering features. This may need cooperation between the software and the hardware. From the viewpoint of software architecture design, the system should have the capabilities of fault detection, fault anticipation and fault termination. For a pilot prototyping robot system, it is very important to have fail-safe features.

2.4. Cost-effectiveness

A sophisticated robot control system is usually implemented with a high-performance computer system. Some control systems for research robots have adopted a multiprocessor VME-based computing platform.[2] Others, although using PCs for hosting high-level control applications, have still kept a VME-based system[4] or a Multi-bus system[1] for low-level control such as data acquisition and joint motor control. Control systems for commercially-available robots, on the other hand, usually are not constructed to have an open architecture, and only offer very limited system expandability. Expanding such systems requires a lot of effort in reverse engineering even with the close cooperation with the original robot manufacturer.[12]

Although cost-effectiveness requirement is an issue related to implementation, it should be addressed from the very beginning of the system design. The system architecture should ensure high portability of the final implementation. In other words, it should be easily realizable on different computer platforms, including the PC ISA/PCI architecture. Another issue related to cost-effectiveness is that the system architecture should be easily developed, debugged, tested and maintained.

3. ANALYSIS AND MODELING

This section describes the analysis and modeling of the multi-robot control system. The modularized modeling serves as the basis and the rationale for the architectural design and the detailed implementation to achieve system scalability and adaptability.

3.1. Function analysis

A robot control system can be modeled with function groups, or sub-controllers, in three stages: the Decision Stage, the Strategy Stage and the Execution Stage. The function model of a typical robotic application in the control perspective is shown in the Figure 3.1. The arrows show the information flows among the subsystems.

3.1.1. Decision-stage controller. The basic function units in the Decision-Stage Controller are Recognition, System Configuration and Task Management. The task goal is interpreted by the Recognition unit and transferred to the System Configuration unit. The Recognition unit also translates the environment information into an environment model. With the knowledge of the capabilities of all the sub-systems,

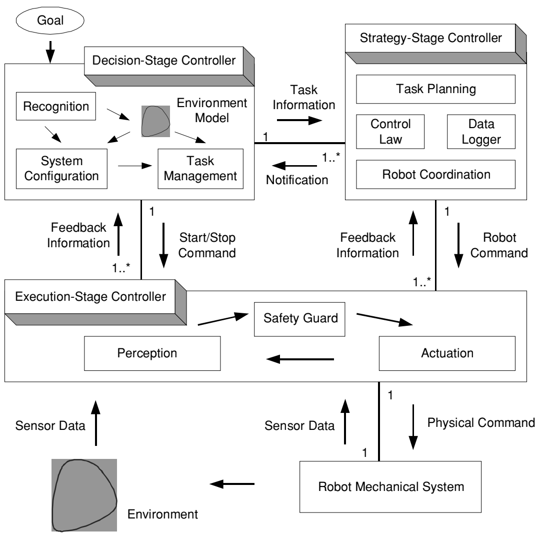


Fig. 3.1. Function model of a robot control system.

the environment model and the goal, the System Configuration unit re-arranges the robot system and the work-cell in order to accommodate the new task. This kind of re-arrangement, or reconfiguration, can be performed mechanically, electrically and/or on the computer network. This requires the relevant system components, i.e., hardware and/or software, to be mobile and re-configurable. The Task Management unit generates the task information and sends it to the Strategy-Stage Controller. When a task is being executed, the Task Management monitors the system status. The Decision-Stage Controller also sends the System-On command directly to the Execution-Stage Controller to start the system, and the System-Off command to stop the system when the task is finished, or if there is an emergency stop situation.

It should be noted that in this paper, the description of the Decision-Stage Controller is only given for the sake of completeness in the context of the robot control system. No further work on the Decision-Stage Controller will be described. Nevertheless, one of the goals of our work was to design an architecture that makes it possible to reconfigure the system structure using the Decision-Stage Controller.

3.1.2. Strategy-stage controller. The basic function units in the Strategy-Stage Controller are Task Planning, Control Law, Robot Coordination and Data Logger. The Task Planning unit translates the task information into task segments with a fine granularity, based on which the trajectories to be tracked are generated. The Control Law unit computes the robot commands based on the trajectories, robot feedback and the specific control algorithms, such as the impedance-control algorithm. These robot and joint commands are sent by the Robot Coordination unit to the Execution-Stage Controller. The Strategy-Stage Controller sends notification to the Decision-Stage Controller indicating the system's dynamic status, and the start and termination of the tasks. The Data Logger unit logs the run-time information of the sub-system for post-task analysis.

3.1.3. Execution-stage controller. The basic function units in the Execution-Stage Controller are Actuation, Perception and Safety Guard. The Actuation unit transfers robot

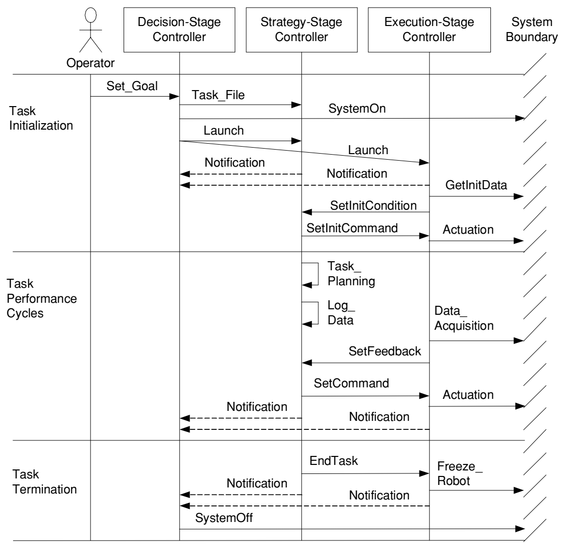


Fig. 3.2. Typical system functional sequences.

commands from the Strategy-Stage Controller into digital values and sends them to the interfaces of the robot joint actuators, which further apply the corresponding physical voltages or currents to the actuators. This unit also accepts the commands from the local Safety Guard unit when there is an emergency situation. The Perception unit acquires the sensor data reflecting the robot dynamic status and the environment status. These raw data from sensor interfaces are further transferred into meaningful information and sent to the Strategy-Stage Controller and Decision-Stage Controller as feedback to the Control Law, Data Logger and Task Management function units. The Safety Guard unit evaluates the robot status and issues an emergency status if necessary. These emergencies include excess in speed limits, joint torque limits, and end-effector force and torque limits. There are other kinds of emergencies not sensed by the sensors. Instead, they are detected by cooperation between the sub-controllers and between the function units (such as loss of communication and/or synchronization).

3.1.4. Functional sequence. The system functionality of a modern robot application is achieved by collaboration of the sub-systems running on a computer or computer network(s). The sequence diagram in Figure 3.2 shows the typical interactions among sub-controllers. There are three different periods during an application task procedure: initialization, task performance and termination. For the two dimensions in the sequence diagram, the vertical dimension represents time, and the horizontal one represents different objects.

The procedure of a robot application is originated by the Task Goal. After setting the Task Information (contained in the Task File) for the Strategy-Stage Controller (SSC), the Decision-Stage Controller (DSC) issues a System-On command to power on the robotic system hardware. Then the DSC launches the SSC and the Execution-Stage Controller (ESC). The notifications from the SSC and the ESC indicate success of the launches. Finally, the ESC gets the initial sensor data and sends them to the SSC; the SSC sets initial robot commands for the ESC to execute. This is the end of task initialization.

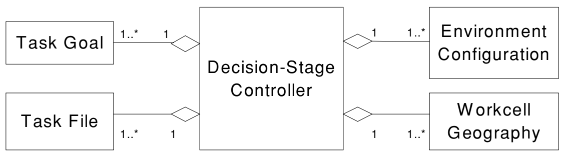


Fig. 3.3. DSC-related data modules.

During task performance cycles, the SSC and the ESC perform their functions until the task is accomplished. The task termination period begins with the End-Task command issued by the SSC. The ESC then stops the robot(s) that it directly interfaces to. After receiving notifications from the SSC and the ESC, the DSC sends a System-Off command to power off the robotic system hardware.

Please note that the termination period shown in Figure 3.2 is the one with a successfully fulfilled task session. A good system design should also deal with the interruption of the task session with unanticipated situations. Sections 4 and 5 discuss in detail the abnormal terminations of a task session.

3.2. Data modeling

In this section, we present the data modules and the relationship among them and the sub-controllers. Then we examine the persistency of the data modules with respect to task sessions. Data modules are abstracted from the physical function model of a robot application (such as that in Figure 3.1). Therefore, it will be clearer if we discuss the data modules and their associations with the sub-controllers.

3.2.1. DSC-associated data modules. As shown in Figure 3.3, four data modules relate to the DSC. The multiplicity is depicted along with the aggregation associations. There is usually only one DSC in a robot application with one or more robot groups, each performing its own task. A DSC can contain all four modules, each with one or more instances. One set of the four modules corresponds to a robot group.

The Task Goal is the information of what is to be done, while the Task File contains the sub-tasks specifying how to fulfill the Task Goal. The Environment Configuration contains different potential work-cell layouts and the subsystem capability information for re-configuring the robot system when necessary. These re-configurations may involve software and hardware components of the control system and the mechanical sub-systems. Work-cell Geography contains the dynamic information of the relative positions and motions of the robots and objects in the work-cell. Typically, this information is acquired by range sensors, vision sensors or human operators when a part of the DSC. The DSC uses this information to evaluate and anticipate overall system status. One of the uses of the Work-cell Geography information for the DSC is to determine whether the system is in a safe condition.

3.2.2. SSC-associated data modules. Figure 3.4 shows the data modules related to the SSC. The SSC usually exists for a robot group performing one task in an application session. Therefore, there is only one instance of the Task File, Work-cell Configuration and Work-cell Geography. Because a robot group may contain more than one robot, the SSC can contain one or more instances of the Robot

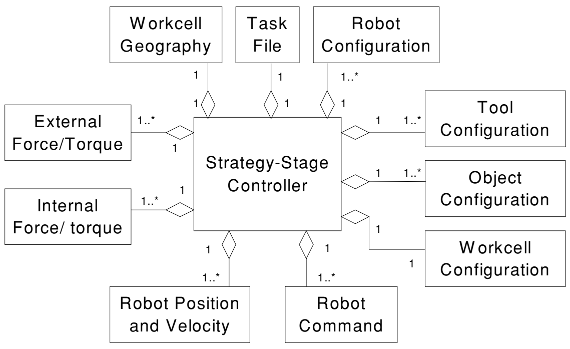
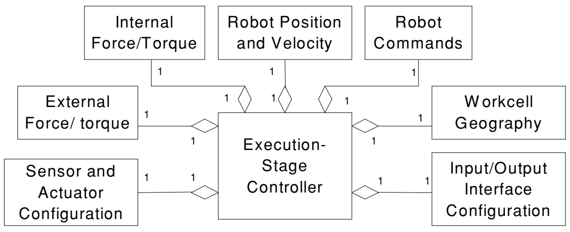


Fig. 3.4. SSC-related data modules.



Internal Force/Torque
Robot Position and Velocity
Robot Commands
External Force/ torque
Execution-Stage Controller
Workcell Geography
Sensor and Actuator Configuration
Input/Output Interface Configuration
1

Fig. 3.5. ESC-related data modules.

Table 3.I. Characteristics of the data modules.

Data modules	Related sub-controller DSC	SSC	ESC	Data persistency
Task Goal	Yes			Static
Task File	Yes			Static
Environment Config.	Yes			Static
Work-cell Geography		Yes		Dynamic
Robot Configuration		Yes		Static
Tool Configuration		Yes		Static
Object Configuration		Yes		Static
Work-cell Config.		Yes		Static
Robot Command			Yes	Dynamic
Robot Position/Velocity			Yes	Dynamic
Internal Force/Torque			Yes	Dynamic
External Force/Torque			Yes	Dynamic
I/O Interface Config.			Yes	Static
Sensor/Actuator Config.			Yes	Static

Configuration, Tool Configuration, Object Configuration, Robot Command, Robot Position and Velocity, Internal and External Force/Torque.

The Task File contains sub-task information, such as target robot position and orientation in position-control mode, or target force and torque in force-control mode, control algorithm related parameters and task end conditions for all sub-tasks. The Work-cell Configuration contains static work-cell information such as the reference of the world co-ordinates, and the positions and orientations of the robot base coordinates. The Work-cell Geography is denotes feedback from the ESC containing dynamic work-cell information. When used by the SSC, it is usual for robots to find an optimal trajectory for online collision avoidance, or to catch a moving object. The Robot Configuration contains geometric measures and the inertia data of the robots in the group. The Tool Configuration and Object Configuration contain the geometry of the tools attached to the robots, and the geometry of the objects manipulated by the robots. The Robot Position and Velocity, and the Internal and External Force/Torque denote the dynamic feedback information of the robots from the ESC. The Robots Commands are the outputs sent to the ESC.

3.2.3. ESC-associated data modules. Figure 3.5 shows the data modules related to the ESC. One ESC is usually associated with each mechanical robotic system. Therefore, the ESC has only one set of Robot Command data and robot feedback: Robot Position and Velocity, Internal and External Force/Torque, and Work-cell Geography. In order to interface to input/output hardware modules, the ESC needs one instance of Input/Output Interface Configuration, and Sensor and Actuator Configuration.

The Input/Output Interface Configuration data contains the addresses and channels of the interface modules, and input/output parameters. The input parameters are used for translating the digital values from input modules (e.g., analog to digit converters, optical encoders) to feedback information. The output parameters are used for translating robot commands to digital values that are provided to the output modules (e.g., digit to analog converters). Similarly, the Sensor and Actuator Configuration information contains the parameters for translation between electrical signals at the interface module ends and the robot status (e.g., force and position) at the sensor/actuator ends.

3.2.4. Data association analysis. Data modules have different characteristics and can be classified as static data (constant data during a task session) and dynamic data (variable as the system status changes). Some modules are owned by one sub-controller, while others are shared by multiple sub-controllers. Table 3.I shows the characteristics of the data modules. Note that the data persistency is with respect to a task session. We can see that all the static data modules (except the Task File) are exclusively related to one sub-controller, while all dynamic data modules and the Task File are shared by multiple sub-controllers. The differentiation between local data and shared data forms the basis of the modular design for the system components and the information exchange scheme.

3.3. Data awareness of function groups

In the previous sections, we have described function modules, data modules and their associations. For these shared data, we can see that they are actually the exchanged information between sub-controllers. In fact, they are shared only in the space dimension (physical memory space). In the time dimension, on the other hand, they are transferred among the sub-controllers. The Task File, for instance, is produced by the DSC and consumed by the SSC. The Robot Command data are produced by the SSC and passed to the ESC to control the robots. From this point of view, the sub-controllers have a producer-consumer relationship.

Further, we have clearly partitioned the sub-controllers (function groups) and bestowed data awareness on them. That is, the sub-controllers know what the prerequisite data are, what the responsibility is, and what the consequent data to be

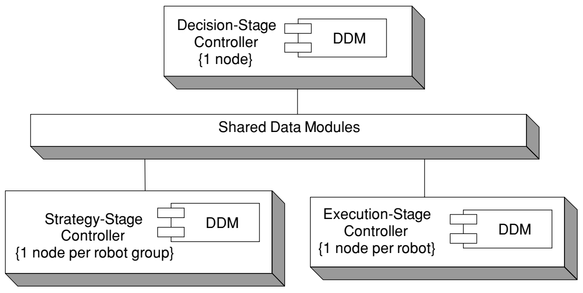

Decision-Stage Controller {1 node}
DDM
Shared Data Modules
Strategy-Stage Controller {1 node per robot group}
DDM
Execution-Stage Controller {1 node per robot}
DDM

Fig. 4.1. Global system structure.

disposed of are. With the data awareness, a sub-controller or a function unit can exist and work well if its prerequisite data is available and the consequent data is disposable. In other words, it is not necessary that the sub-controllers or the function units have global knowledge of the robot system. Only having local knowledge of the robot or robot group under the control of the sub-controllers is sufficient for the survival of a sub-controller or a function unit. This kind of data awareness improves the modularity and mobility of system components and hence system scalability and adaptability.

4. ARCHITECTURE AND DESIGN

This section presents the architectural design of the multi-robot control system based on the analysis and modeling in Section 3. We adopt a top-down style by starting with the global system structure, then the sub-controller structure, and finally the structure of the individual function units, and run through these topics with data related concepts: data connection, data isolation and data encapsulation. We conclude this section with a trade-off analysis of the system design by examining the design against the design requirements discussed in Section 2. The discussion of the design of the Dynamic Data Management and the data modules is deferred to the next section, since most of their features more or less depend on implementation-related issues.

4.1. Global system structure

The global system structure provides a bird's-eye view of the whole system when running an application task. In order to meet the design requirements in Section 2, we must construct loosely coupled system components. Based on the system modeling and analysis in Section 3, one way to achieve a loosely coupled connection is to build up the system components functionally independent of each other. The connections among the system components are realized by the shared data modules, which the sub-controllers treat as data repository. From the perspective of the sub-controllers, the peripheral world is only the data source and destination. Therefore, they can home in to wherever the data repository is available. It is worthwhile noticing that for a real-time system, there are temporal requirements for the availability of the data repository, which will be discussed in Section 5. Figure 4.1 shows the system structure.

4.1.1. Data connection. Since the goal is that there are no direct functional interactions among sub-controllers, we need Dynamic Data Management (DDM) modules to update the shared data when needed. Those shared data are actually input and output information exchanged between the sub-controllers. The dynamic data updating by the DDM accomplishes the data exchange between the sub-controllers and closes up the information flow loop. In order to achieve a modular system, the sub-controllers should have a uniform interface to get access to the shared information, especially for dynamic data. The DDM realizes the shared data object, to which the sub-controllers will conform. Since potentially the sub-controllers may physically be hosted on different machines, each sub-controller needs to have its own DDM so that it can be mobile and achieve system scalability and adaptability. The structure of the sub-controllers will be discussed in the next section.

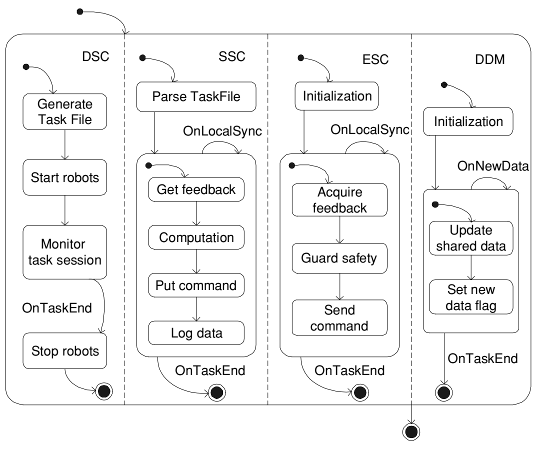


Fig. 4.2. Global system behavior.

4.1.2. System behavior. The state-chart diagram in Figure 4.2 shows the designed global system behavior. We can see that all the sub-controllers and DDM are running as independent processes. We use orthogonal states to depict the independence among the subsystems.

A multi-robot control system is a real-time synchronous system. An application task needs the cooperation of different parts of the system by some means of synchronization. While using local synchronization does not cause problems for the loose coupling of the system, using an explicit global timing signal, however, brings the system back to a tightly coupled one. Moreover, explicit global synchronization requires more system resources or perhaps a more powerful hardware system than that without explicit global synchronization. This is even more true when the system is implemented over a network. In Figure 4.2, we can see that there is no explicit global synchronization and all the processes are running under their own contexts. We realize the global synchronization implicitly through data dependency, which will be discussed in later sections. Although all the processes have time-critical characteristics, only the SSC and the ESC need synchronization on every sampling and control period. The DSC needs only to react quickly to emergencies. The DDM needs no synchronization at all, not only because the data updating occurs only when needed, but also because different data could change at different rates. That is the other positive effect of local synchronization: the sub-controllers

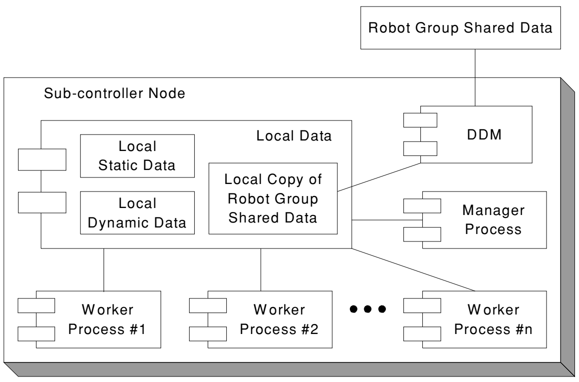


Fig. 4.3. Sub-controller structure.

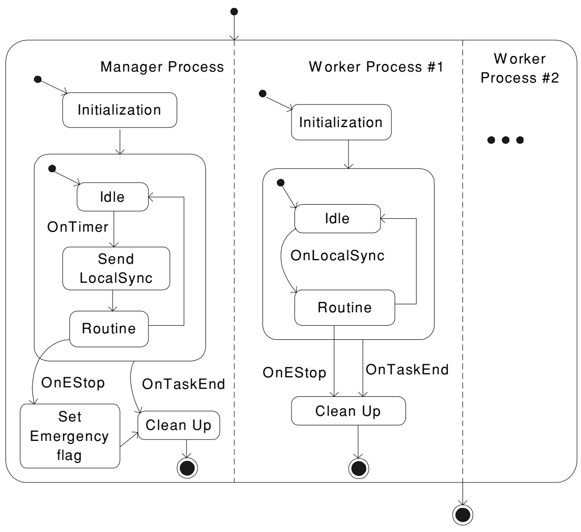


Fig. 4.4. Sub-controller behavior.

can run at different frequencies, which is particularly useful for a heterogeneous system.

4.2. Sub-controller structure

In this section, we discuss the structures of the SSC and the ESC, while leaving the DSC sub-system open for future work. The design and implementation of the DSC are beyond the scope of this paper.

4.2.1. Static structure. Figure 4.3 shows the detailed sub-controller structure. A sub-controller contains one Manager process and one or more Worker processes. The basic responsibility of the Manager process is initializing the controller group, launching Worker processes, synchronizing the controller group and monitoring run-time status. The Worker processes are responsible for specialized functions of the sub-controllers. For an ESC, for example, there are worker processes for Data Acquisition, Command Sender and Safety Guard.

The connection between the SSC and each ESC is the shared data object exchanged by the DDM via the Robot Group Shared Data object. The sub-controller gets access only to the local data module for the input and output information, without any awareness of the existence of the other sub-controller(s). In other words, the local data module is the whole interface of the sub-controller for information exchange.

The partitioning of the controller's function into separate processes is based on the following conditions and concerns:

(i) It can be naturally partitioned into independent function units;
(ii) There is little data dependence between the function units, so that they can be running independently;
(iii) It is necessary and possible to prioritize and schedule the function units separately;
(iv) The separation and combination can improve system safety.

4.2.2. Data isolation. From Figure 4.3, we can see that in the local data module, there are extra local dynamic data and a local copy of robot group shared data, in addition to the basic static data (as shown in Table 3.1). Local dynamic data are used for maintaining system integrity and safety, while a local copy of robot group shared data is used for improving real-time performance. The latter is examined next.

Although all the controller processes are running autonomously, or asynchronously, the sequence of data production and consumption, have to be synchronized in a specific order, especially in a control system. That means the processes will get access to the global data at different instants and this will cost a lot in terms of real-time performance. By keeping a local copy of the global shared data, which is updated by the DDM only once for each sampling/control period, the need for access to the global shared data object is significantly reduced. The real-time performance is hence improved.

The shared data object acts as the connection among the sub-controllers as discussed above. That is because of the data dependency between the sub-controllers. Within the scope of a sub-controller, the controller processes are constructed for the least dependency as was described in Section 4.2.1. The local shared data object actually acts as the isolation among the controller processes and serves as a medium for them to get access to the robot group shared data. This is an extension of the data connection among the sub-controllers in that the controller processes can run with significant autonomy, while fulfilling naturally tightly coupled functions for a robot task session.

4.2.3. Sub-controller behavior. Figure 4.4 shows the sub-controller behavior. The Manager process and all the Worker processes run under their own contexts, which are represented by the orthogonal states. All Worker processes have the same behavior, but different routines. There is an independent timer for each sub-controller. The Manager sends the local synchronization signal with a predefined frequency of the local timer. It is not necessary that all the sub-controllers run at the same frequency, since the sub-controllers synchronize themselves independently.

When the task is completed successfully, the SSC sets the Task End flag. Within the scope of the sub-controller, the Manager issues the Session End signal, on which all the Worker processes will stop running and quit. All the processes can set an Emergency Stop flag if they detect any unsafe situations. Once the Emergency Stop flag is sensed, the Manager sends the Session End signal to the

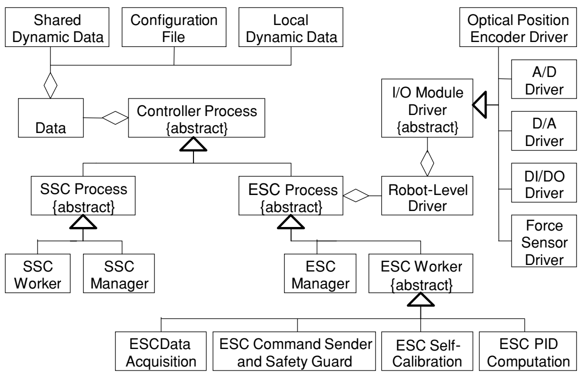


Fig. 4.5. Controller process structure.

local Workers and sets the Emergency Stop flag in the local copy of Robot Group Shared Data as well. Finally, the DDM propagates the flag to all the related sub-controllers.

4.3. Controller process structure

The controller processes are the finest granules of the independent run-time entities. They have different responsibilities, such as the Manager and the Workers, and are specialized for various functions, such as control computations, data acquisition, sending commands, etc. However, they have the same static structure and dynamic behaviors, as those designed in this paper.

4.3.1. Static hierarchy. The class diagram in Figure 4.5 shows the static structure and inheritance hierarchy of the controller process. On top of the hierarchy is the Controller Process (an abstract class) that contains a Local Data module. The Local Data module contains Shared Dynamic Data (local copy of the robot group shared data), Local Dynamic Data and Configuration Files. The contents of these data objects can be found in Section 3.

The SSC Process and the ESC Process are generalized from the Controller Process. The difference between the SSC Process and the ESC Process is hat the latter has a Robot-level Driver in the fact that it needs to communicate directly with the robot control system hardware. The SSC Manager and SSC Worker are the sub-classes of the SSC Process. The SSC Manager has a timer for timing-based synchronization, while the SSC Worker has not, since it only needs to receive the synchronization signal. This is the same situation in the ESC: only the ESC manager has the timer. The ESC Worker is further generalized to four specialized work processes. Those processes are ESC Data Acquisition, ESC Command Sender and Safety Guard, ESC Self-calibration, and ESC PID Computation.

The Robot-level Driver contains an I/O Module Driver, which interfaces directly to the input/output boards. The I/O Module Driver is a set of hardware-specific and general-purpose classes/functions handling raw data, which is understood directly by the input/output boards. The Robot-level Driver is actually a wrapper library that serves an isolation layer between high-level software and the low-level hardware input/output operations. It presents a virtual machine to the high-level software and maps it to the I/O hardware modules. To achieve this, the Robot-level Driver interprets the meaningful robot-level commands to board-level ones and executes them with the relevant I/O Module Driver operations. With this structure, the low-level hardware operations are transparent to the high-level application software. A properly designed protocol between the Robot-level Driver and I/O Module Driver will achieve the isolation between high-level software and low-level hardware. The Robot-level Driver can adapt to any line of input/output boards and the I/O Module driver can adapt to different robotic systems. This will facilitate system reconfiguration.

4.3.2. Data encapsulation. From Figure 4.5, we can see that every individual controller process encapsulates all relevant data modules and objects. All the controller processes treat the static and the dynamic data in the same way, taking advantage of the fact that the shared dynamic data will be updated by the DDM every sampling and control period. This kind of data encapsulation makes it possible for all the controller processes to work in a self-sufficient manner, while the DDM modules ensure data dependency, i.e., data producer-consumer relationship among the controller processes.

With the object and data module encapsulation, realizations of some important functions in a multi-processing system become as easy as that in a single-process system. For example, when the robot system starts, there is an uncertain period between the power-on of the robots and the first issuing of a robot command. We need to freeze the robots as early as possible to eliminate any possible unspecified robot movement caused by any environment disturbance. In a multi-processing system, the session is started with the Manager process, which in turn launches the specialized Worker processes. The process that has the ability to issue a freezing-robot command might not start in the early stage of the task session, nor has it the priority to execute earlier than the other processes. This might increase the uncertain period. Now the ESC Manager has the Robot-level Driver and the system hardware configuration. It can send the freeze-robot command at the earliest instant before spawning any other Worker processes, even if its normal operation priority is the lowest in the controller group.

4.3.3. Controller process behavior. All controller processes have four mandatory states: Initialization, Routine, Cleanup and Idle as shown in Figure 4.4.

- **Initialization:** entry of the process. This does process initialization before the application session is started.
- **Routine:** body of the process. This does synchronized work during each control and sampling period.
- **Cleanup:** end of the process. This is executed before the actual termination.
- **Idle:** after the Routine is finished and before the next control and sampling period. This does non-synchronized work (if any) during the session.

4.4. Benefits and overheads

The goal of the modular design of the multi-robot control system is to meet the requirements discussed in Section 2. In addition, our aim is to facilitate an efficient implementation that is based on the design. The following summary discusses the tradeoffs of the system architecture and design.

4.4.1. Benefits

a. Scalability and adaptability. All the run-time entities (robot group controllers, sub-controllers and controller processes) are functionally self-sufficient. There is no explicit global synchronization, and the data dependency is ensured by the DDM modules. With the autonomy and mobility of the system components, this system design can adapt to reconfigurations and scale-up by expansions. With the two-layered control hardware driver, the system can also adapt to various robotic systems and different interface hardware. The benefits of scalability and adaptability are gained from the modular design of the system structure and its behavior.

b. Schedulability and predictability. The overall system has been partitioned with least dependency. The system components can be scheduled independently. They run virtually in their own context. Different hardware resources are only communicated with specific run-time entities. For instance, the input hardware modules communicate with ESC Data Acquisition only. The output hardware modules are related to ESC Command Sender and Safety Guard only (except at the very beginning of the task session, the ESC Manager will send the *freeze robot* command). Another example is that within the sub-controller node, only the Manager process can display run-time information. These arrangements in the system behavior design effectively prevent possible priority inversions, if the processes are assigned different priorities. All run-time behaviors of the processes are predictable. This is one of the most important issues when a real-time synchronized task is performed by a group of processes running either independently or concurrently. The benefits of schedulability and predictability are gained mostly from the system behavior design.

c. Safety and reliability. The contribution of the structural and behavioral design to system safety and reliability is mainly gained from the loose coupling of the system components. First, all the components are functionally independent. Second, the software-hardware connections are designed as one-to-one relationships. Third, the robot only gets the next command if its current states pass the safety check at every sampling/control period. The first two design issues prevent the common mode failures, the single-point failures that affect multiple parts.[11] The third will keep the consequence of the system behavior deterministic and restrict the energy released by the system to the environment within bounded limits. The safety issues will be addressed further in Section 5.

d. Cost-effectiveness. With a modular and object-oriented design, the system components can be developed, debugged and tested independently, either by a single programmer or by a team. The individual components can be modified and the behavior can be changed, while keeping the other parts of the system intact. Well-tested modules may be re-used at a future time for the system expansion or for constructing new systems.

4.4.2. Overheads. In order to maintain the integrity and safety of a system with a group of autonomous sub-controllers that may be distributed on a computer network, extra data traffic in the communication is unavoidable. To improve the autonomy and real-time performance of the system, extra copies of data modules are justified. That is to trade space for time. However, with the availability of high-performance low-cost computer systems, these overheads may not affect the system real-time performance as a whole. The benefits gained from the overheads make them worthwhile.

5. IMPLEMENTATION

This section presents the implementation of a multi-robot control system based on the analysis and design presented in the previous sections. The system is built on the QNX RTOS[9,10,13] using standard *C*++. This section also discusses the fail-safe features including fault-detection, fault-anticipation, and fault-termination in the implementation. Although the target system runs under the QNX RTOS, it is implemented with a high degree of portability, since all the important parts of the implementation have been made to conform to the POSIX standard. Although some features of the implementation are aimed at the target robot system, the concept and the practical aspects of the work can be extended to a generic control system with multiple processes.

5.1. Prioritizing and scheduling

With real-time systems, although it is important that the operating system should quickly respond to external events, the most crucial point is how deterministic those responses are in terms of the instants when they occur. The more deterministic the operating system, the more suitable that system will be for real-time applications. A typical robotic application has sampling/control frequency ranging from 50 Hz to 500 Hz; the corresponding period ranges from 20 ms to 2 ms, as is the case in this research. The real-time performance of QNX is quite suitable for such an application with a reasonable number of processes.[13] Under the QNX system, every process has a priority. A higher-priority process can preempt lower-priority processes. The system schedules the process with the highest priority to run when the process is ready. QNX provides three scheduling methods: FIFO (First-In-First-Out), round-robin, and adaptive scheduling. While these methods are effective on a per-process basis, they affect the behavior of a system with a group of processes interacting with each other.

From the system design described in Section 4, we know that the sub-controllers work in an autonomous mode. There is no necessity to schedule the sub-controllers by means of the QNX facility. However, the controller processes are synchronized within the scope of every sub-controller. Table 5.I shows the priorities assigned to the processes in the implementation. In QNX, the priorities range from 0 (the lowest) to 31 (the highest). Processes started by the Shell normally have a priority of 10, which is inherited from the Shell by default. We choose the lowest priority of the controller process as 15 to make the robot controller run at a higher priority than the Shell commands and their child

Table 5.I. Controller process priorities.

Priority	ESC node	SSC node
19	ESC Data Acquisition	
18	ESC Command Sender & Safety Guard	SSC Worker
17	ESC Self-calibration	
16	ESC PID Computation	
15	ESC Manager	SSC Manager

processes. We use round-robin as the scheduling method. If the ESC and SSC run on different computer nodes, the scheduling method does not actually affect the system behavior, since the processes run with different priorities. When the ESC and SSC run on the same computer, the processes with the same priority will be scheduled for a time-slice in turn.

For a group of prioritized processes, it is important to prevent any scenarios of priority inversion. Priority inversion is the situation that a higher-priority process is blocked by a lower-priority process. This is usually caused by resource conflicts when the processes share system resources. For example, a higher-priority process cannot run when the required resource is owned by a lower-priority process. From the design in Section 4, we can see that the hardware resource sharing (such as input/output modules, hard disk drive, and the display monitor) is avoided during the entire task session. There are no possibilities of priority inversion, either when all the controller processes are well conducted or in a fault situation.

5.2. *Information sharing and exchanging*

Information sharing and exchanging forms the cornerstone of the system architecture. There are two types of shared data modules for the information exchange. One is the Local Data that is shared among the controller processes within the scope of a sub-controller. The other is the Robot Group Shared Data that is exchanged among an SSC and all the ESCs in a robot group.

5.2.1. Structural implementation. As shown in Figure 5.1, the Robot Group Shared Data contains robot commands and feedback data, which are grouped on an individual robot basis for the sake of efficiency. Robot Command Data contains robot joint motion/torque commands, joint on/off commands and session status. Session status is issued by the SSC Manager and indicates whether the session is in-progress or finished. Feedback Data contains joint position and velocity, end-effector force and torque, joint torque, and emergency stop status. The emergency stop status indicates the emergency stop flag of the related robot.

Figure 5.2 shows the structural implementation of a Local Data module. The information is classified into Local Static Data, Local Dynamic Data and local copy of Robot Group Shared Data. The static data contains system configurations and the run-time address of the module. In the dynamic data category, Mutual Exclusion Status is used for data consistency by ensuring the data will be read or written by only one process at any instant. Synchronization Flags are

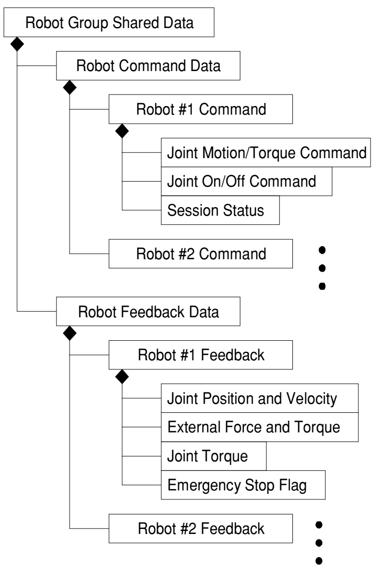


Fig. 5.1. Structural implementation of robot group shared data.

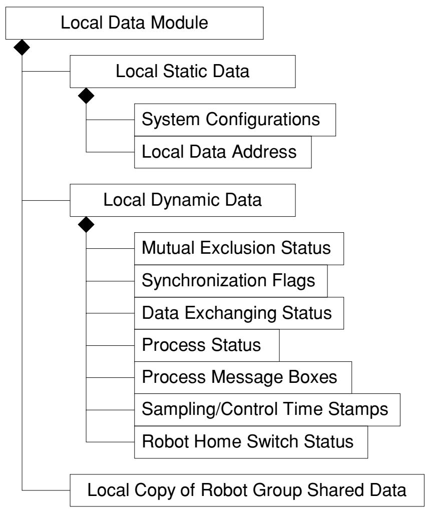


Fig. 5.2. Structural implementation of a local data module.

used to ensure the order of data production and consumption. Data Exchange Status states the freshness of the data from the SSC. Process Status indicates whether the related process is alive or not. The Process Message Boxes are used for the worker processes sending messages to the manager process for displaying or tracing run-time status. Sampling/Control Time Stamps are the time instants at which the feedback is sensed and the robot command is sent to the robot. The Robot Home Switch Status indicates whether the robot joints have reached the joint home switches or not, which is mainly used for automatic calibration. The Local Copy of Robot Group Shared Data contains the same information as that in the Robot Group Shared Data. It is used as the data repository of the controller processes, and is updated and transferred by the DDM.

5.2.2. Physical implementation. One of the design goals is to achieve a loosely coupled running mode among the sub-controllers and the controller processes. Therefore, we implemented the Local Data module as a POSIX-conforming

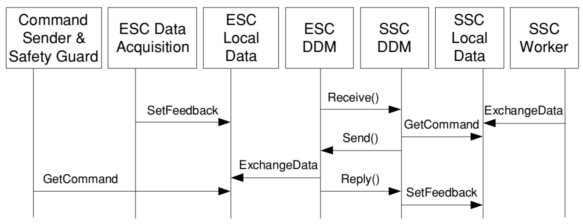


Fig. 5.3. A typical round of data exchange with QNX message passing.

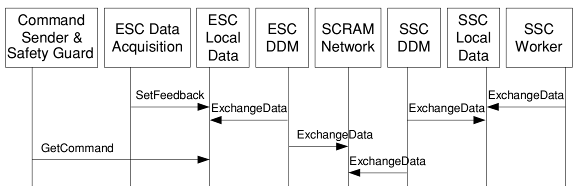


Fig. 5.4. Data exchange with RTCI/SCRAM Network.

shared memory, which is supported by the QNX RTOS. The DDM sitting on each sub-controller node is responsible for transferring and updating the Robot Group Shared Data.

There are two physical implementations of the DDM and Robot Group Shared Data. The first uses QNX message passing between the SSC DDM process and the ESC DDM process. Figure 5.3 shows a typical round of the data exchange in a sampling and control period. At the beginning of each data exchange, the ESC DDM calls the *receive* () function and becomes blocked waiting for the data from the SSC DDM. Then the ESC Data Acquisition process sets the feedback information to the ESC Local Data, and the SSC Worker exchanges data with the SSC Local Data (gets feedback and sets command). After getting robot commands, the SSC DDM sends them to the ESC DDM and waits for a reply. The ESC DDM returns from the receive-blocked state once it gets the command from the SSC DDM, and exchanges data with the ESC Local Data (sets command and gets feedback). The ESC DDM then replies to the SSC DDM with the feedback data. At this point the SSC DDM returns with feedback data and sets the data to the SSC Local Data module, which will be used for the next sampling/control period. Once the robot commands are ready, the ESC Command Sender and Safety Guard process sends them to the robot hardware.

The second physical implementation of the DDM and Robot Group Shared Data makes use of the Shared Common RAM Network (SCRAM Network), which is a commercially available product.[14] Any time an application process updates data located in its local SCRAM Network memory, the address and data are broadcast immediately and automatically to all other nodes on the network. Figure 5.4 shows a typical round of the data exchange in a sampling and control period with the SCRAM Network. The direct QNX message passing is replaced by exchanging data with SCRAM Network by the ESC DDM and the SSC DDM. So, there are not any more blocked states (receive-blocked, reply-blocked and send-blocked states) for the controller processes.

5.3. *Synchronization*

There are two types of synchronization: time-based synchronization and datum-based synchronization. The time-based one is only used within the scope of a sub-controller node. The datum-based synchronization is used for both intra-sub-controller data consistency and inter-sub-controller data dependency. The latter is used as an alternative to direct temporal synchronization among the sub-controllers.

a. Temporal synchronization. A discrete control system fulfills the application task on a discrete sampling/control basis. All the controller processes execute their routines during every sampling and control period, and hence need to be synchronized. The temporal synchronization is implemented by using POSIX signals supported by QNX RTOS. Armed with a QNX real-time timer, the Manager processes send the sync signal to all worker process within the sub-controller node on a pre-defined frequency. The termination of the task session is also implemented as a synchronized event, during which the Manager sends the task end signal to the Worker processes. The Worker process is constructed to be able to receive those signals only.

b. Datum-based synchronization. Datum-based synchronization is implemented by making use of POSIX semaphores, which are also supported by the QNX operating system. Two important features of semaphores make them the ideal candidates for the means of synchronization for data dependency and data consistency (mutual exclusion). The first is that the operations of semaphores (i.e. “post” and “wait”) are guaranteed atomic. That is, the operating system makes sure that they are not interruptible. This is extremely useful for mutual exclusion of the processes having access to a set of shared data. The second is that semaphores are “async safe” and can be manipulated by signal handlers. Therefore, we can use signal handlers in different processes to transfer synchronization information to each other.

c. Data dependency. Under the system architecture presented in Section 4, the sub-controllers run in an autonomous mode. However, data dependency among all parts of the robot control system still needs to be maintained in order to accomplish the application task. For instance, the SSC computes robot commands based on the robot feedback information acquired by the ESC. This means that the SSC cannot start to work before the ESC Data Acquisition process is launched when the task session starts. Figure 5.5 shows the data dependency during the starting of a task session. We can see that no matter whether the ESC starts first or the SSC starts first, the ESC Data Acquisition and the SSC DDM will meet a synchronization point. This point ensures that the SSC Manager is not launched until all the ESC Data Acquisition processes get the initial states of all the robots in the group. The subsequent robot commands are computed from the correct robot states. Otherwise, the robots could run under the commands based on uncertain states, which may result in unpredictable behavior of the system.

d. Data consistency. Within the scope of an individual sub-controller node, all the controller processes get access to the shared Local Data module for information exchange. Should the data be changed by one process while it is read by another

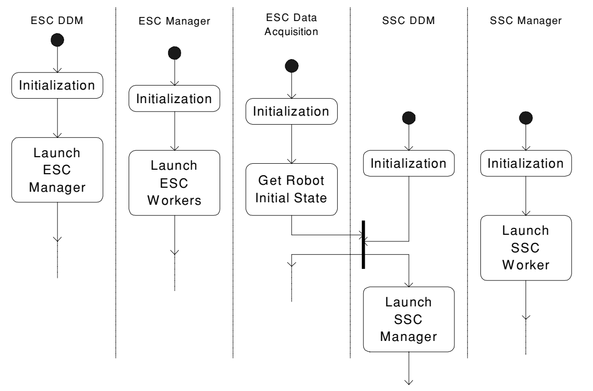


Fig. 5.5. Starting of an application task session.

process, the reading process cannot get the correct data. We adopt the POSIX semaphores to guard the accesses to the shared data. Any time a process wants to exchange data, it will wait for the semaphore. The process reads and/or writes only when the data is not locked. Once a process gets access to the data, the data is locked to the process. Before the process gets back, it posts the semaphore to unlock the data. If a process with a higher priority is waiting for a semaphore, it is blocked and gives up the CPU. The operating system can then schedule the process with lower priority to run. This prevents priority inversion. Since the semaphore-wait and semaphore-post are atomic operations, the shared data can only be in either locked or unlocked state, but not in transition between the two states.

5.4. *Run-time safety*

Run-time safety is usually achieved by incorporating fault-tolerance and fault recovery facilities in the system. These may need combinations of software and hardware features. However, we will only focus on the software ability to achieve fault detection and failsafe operation.

5.4.1. Fault detection. There are three data facilities for fault detection. They are process status, process message boxes and data exchange status, all of which are located in the Local Data module on each sub-controller node (please refer to Figure 5.2).

a. Process status. This status is used for detecting errors within the scope of a sub-controller node. During every sampling/control period, the Worker processes will sign up their status into the Process Status in shared Local Data module. The process status is then checked and reset by the Manager process. If for some reason, a Worker process fails to sign-up, this will be deemed as an occurrence of an error.

b. Process message boxes. There is a message box allocated for each process, which is for the process to put descriptive messages into if any irregular situations are detected by the process. The Manager process will display the irregular message on the monitor to bring it to the operator's attention. The message boxes are transferred within the sub-controller node.

c. Data exchange status. This status is used to detect any errors of the data exchange between sub-controllers. During a task session, if the communication between sub-controller nodes is interrupted for some reason, the ESC will not get the robot commands feeding from the SSC. If this situation lasts for a certain amount of time, an error flag is set up. This scheme is implemented by means of a starving index. Every time after the ESC Command Sender and Safety Guard process sends a set of commands to the robot, the process increases the starving index. Every time the ESC DDM gets a set of robot commands, the DDM decreases the starving index. If the starving index exceeds a pre-defined limit, an error is deemed to have occurred. This implementation can accommodate sub-controller nodes running under different frequency.

5.4.2. Fault anticipation. Fault anticipation is implemented by using the ESC Command Sender and Safety Guard to check robot joint velocities, end-effector force and joint torques before sending the robot commands to the robot for execution. If the magnitudes of these variables exceed pre-defined maximum values, an error flag is set to indicate that the system may have a fault. While the values beyond the limits may not result from a fault situation, they could bring the system into states that are out of control. From this point of view, this scheme is a kind of fault anticipation. This scheme is especially useful for a prototyping system.

5.4.3. Fault terminations. We have planned for the fault termination scheme to achieve failsafe operation. The failure may result from either a system fault or a run-time error. While every controller can set an emergency stop flag, only the Manager process can start a fault termination procedure. First, the robot in question is frozen immediately, either by the ESC Command Sender and Safety process or by the ESC Manager process, whichever occurs earlier. Then, the ESC Manager sets the emergency stop flag, which is transferred by the ESC DDM to the SSC DDM. Third, the SSC DDM sends the flag to all the ESCs in the robot group. Finally, every Manager process terminates all the Worker processes in its controller node by sending the termination signal to all Worker processes. The three fault terminations are as follows:

a. Unsafe termination. During each control period, if the joint velocities, Cartesian force and torque, and joint torques do not pass the safety checks, the ESC Command Sender and Safety Guard process will freeze the robot immediately. The ESC Manager then starts an emergency stop procedure.

b. Starving termination. During the task session, if the starving index exceeds the limit, the ESC Manager freezes the robot under control and starts an emergency stop procedure.

c. Abnormal termination. Any other unexpected run-time error will cause abnormal termination of a session.

An ESC process or the human operator freezes the robot and an emergency stop procedure starts.

6. EXPERIMENTAL VALIDATION

This section describes experimental tasks and the results from the tests running on a test-bed with dual seven degrees-of-freedom (DOFs) manipulators. The purpose of

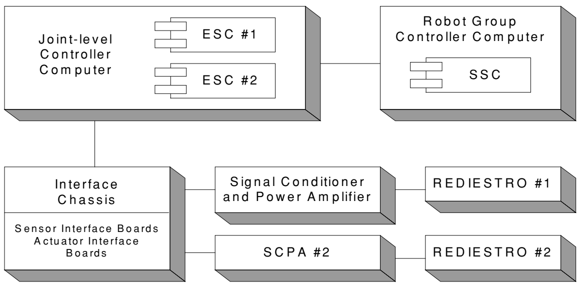

Joint-level Controller Computer
ESC #1
ESC #2
Robot Group Controller Computer
SSC
Interface Chassis
Sensor Interface Boards Actuator Interface Boards
Signal Conditioner and Power Amplifier
SCPA #2
REDIESTRO #1
REDIESTRO #2

Fig. 6.1. The REDIESTRO controller hardware system.

the experiments is to examine and validate the effectiveness of the design and implementation.

6.1. The robotic system

The test-bed consists of two 7-DOF redundant manipulators (REDIESTROs – REdundant, Dexterous Isotropically Enhanced Seven Turning-pair RObots). Each robot was designed to be kinematically isotropic in addition to being redundant.[15–17] The goal was to build an advanced highly dexterous dual-arm robotic manipulation environment that can be used for developing appropriate methodologies, and hardware and software testing. The capabilities of the environment are trajectory planning, position control, force control, impedance control, object contact, collision avoidance, dual-arm open-chain and closed-chain control, and tele-operation. The two manipulators are kinematically mirror images of each other. The actuator on each joint of the two robots is a DC motor with a harmonic drive system. All actuators on REDIESTRO 2 are equipped with incremental optical encoders, tachometers and torque sensors, while the actuators on REDIESTRO 1 have encoders only. All REDIESTRO 1 joints have brakes, but only the second, third, fourth and fifth joints on REDIESTRO 2 have brakes. All the joints of both robots have Hall-effect sensors as the home switches, and both robots have six-axis force sensors mounted on the last link.

The home switches are used for the control system to get the reference positions of the joints, since the incremental encoders can only measure relative changes of joint positions. As part of its implementation, the control system gets the initial joint positions from a file instead of the robots themselves. When a task session starts, the counters of the encoder interfaces have to be initialized with the recorded joint positions in order that the data read from the encoder counters reflects the absolute joint positions. Accordingly, when the session ends, the controller records the final joint positions in a file.

Because of joint flexibility, the manipulators are difficult to calibrate kinematically. Also, since not all the joints of REDIEDTRO 2 have brakes, there may be a loss of configuration due to disturbances in the work-cell. A self-calibration process therefore needs to be constructed that will run whenever there is need to get back the reference position of the robots. Another issue concerns run-time safety arising from the lack of the brakes on the first, sixth and seventh joints of REDIESTRO 2. The controller must be proactive and stop the robots if there is any suspected fault. The robots should not be allowed in any period to have an uncertain state. This is achieved using the fault detection, fault anticipation and fault termination schemes described in Section 5. The goal is that the controller must have the ability to detect any possible fault as early as possible and terminate the task session before control is lost, and hence achieve fail-safe operation.

6.2. Controller hardware

Figure 6.1 shows the hardware system of the dual-arm REDIESTRO controller. The SSC runs on the computer for the robot-group-level controller. The ESC runs on the computer for the joint-level controller. The sensor/actuator interface chassis holds all input/output interface boards. The signal conditioner and power amplifier is wired to the sensors and actuators on the robots.

The input/output boards include

- Encoder interface cards for the optical encoders measuring joint positions,
- Digital-to-analog converter (DAC) cards for DC motor control,
- Analog-to-digital converter (ADC) cards for joint torque sensors and tachometers,
- Digital input/output cards for joint home-position switches (inputs) and brakes (outputs),
- Force sensor receiver cards for the robot end-effector 6-axis force/torque sensors for measuring Cartesian forces and torques.

The joint velocity is actually obtained by digital differentiation of joint position data. There are no sensing facilities to get joint acceleration information. One of the goals of this work was to achieve a cost-effective solution for a sophisticated multi-robot control system. Our aim was to achieve a totally PC-based implementation with all the interface boards running on the PCI/ISA bus system. Table 6.I shows the hardware components for the PC-based dual-arm

Table 6.I. Hardware components of the PC-based dual-arm controller.

Model	Description	QTY	Comment
Keithley DAS-1802	16-channel 16-bit ADC	1	REDIESTRO #2
Keithley DDA-16	16-channel 12-bit DAC	1	REDIESTRO #1, #2
Keithley M5312	4-axis 24-bit Quadrature Encoder Interface	4	
Keithley PIO-24	24-bit Digital I/O	2	
JR3 ISA Receiver	DSP-based Force Sensor Receiver	2	
DELL PC	Pentium 233 MHz	1	
DELL PC	Dual-processor 350 MHz	1	SSC Computer

Table 6.II. Segments of the self-calibration task.

Segment	Sub-task	Goal	Reference
1	Clockwise Move	− Range	Initial joint position
2	Counterclockwise Move	+ Range	Initial joint position
3	Final Positioning	Final Position	Corrected zero positions

controller. All interface boards are plugged in an ISA-bus chassis. The chassis was connected to a PC system through a PCI/ISA bus extender. The robot-group controller computer runs on a dual-processor 350 MHz Pentium II PC. Both the controllers were real-time multi-processing programs running under QNX.

6.3. Task #1: self-calibration

The self-calibration task can be performed by the ESC only, since the task goal is to get joint positions with reference to the zero positions corrected by the home switches. The objective of the experiment was to examine basic real-time performance of the multi-processing control system in joint space. Since there are no coupling operations between each other, the two REDIESTRO manipulators can run self-calibration either at the same time or separately. Moreover, there is no need to use the DDM and Robot Group Shared Data, and only the Local Data module is needed for the data exchange among the ESC Worker processes. The participant processes are the ESC Self-calibration, the ESC PID Computation, the ESC Data Acquisition, the ESC Command Sender & Safety-Guard, and the ESC Manager process.

6.3.1. Task segments. There are three segments in this task for each robot joint. These are Clockwise Move, Counterclockwise Move, and Final Positioning. The robot is in position control mode during all the segments. The duration of all the segments is 10 seconds. Table 6.II lists the sub-task, goal position and the reference of the goal position for each segment. The value of the Range is typically chosen as 10 to 15 degrees. At the end of segment #2, the positions of all the home switches can be found without any exception. The third segment will put all the joints at the desired positions with respect to the corrected zero positions.

6.3.2. Sampling and control frequency. The working frequency was set at 200 Hz. We have tested the system up to 500 Hz with the two robots and a total of 10 processes concurrently running on the single 233 MHz Pentium PC, without any degradation in the real-time performance.

6.3.3. Analysis of the results . The plots in Figures 6.2 to 6.4 show the experimental results consisting of joint positions, joint velocities, and joint torque commands. The joint velocities were obtained by digital differentiation of the joint positions. The final positions of all the joints were specified at the home switch positions. The joint position trajectories were planned using a fifth-order polynomial and the joint torque commands were computed with a PID algorithm. At the end of each of the segments (t = 0 seconds, 10 seconds, 20 seconds, and 30 seconds), the joint velocities were specified to be zero. At the starting point (t = 0 seconds) and the final point (t = 30 seconds), the joint accelerations were also required to be zero. From Figure 6.2, we can see that the robot performed as was specified and expected. Only at t = 10 seconds and t = 20 seconds, there are non-smooth transitions when joint #2 changes its direction, from negative to positive, and back to negative from positive. This is mainly caused by the gravity effect, which had the greatest effect on the motion of joint #2. Figure 6.3 shows the result of differentiating with respect to time the plots in Figure 6.2, both of which formed the basis for computing the command torque that is shown in Figure 6.4. The non-smooth transitions in the joint #2 trajectory are the jumps in the negative direction. There was another jump in the negative direction when the robot begins to move. This is also caused by the effect of gravity at the instant when the brake is released. All these jumps are soon corrected by the commands computed by the PID algorithm.

The plots show that the basic real-time performance of the multi-processing control system, such as inter-process

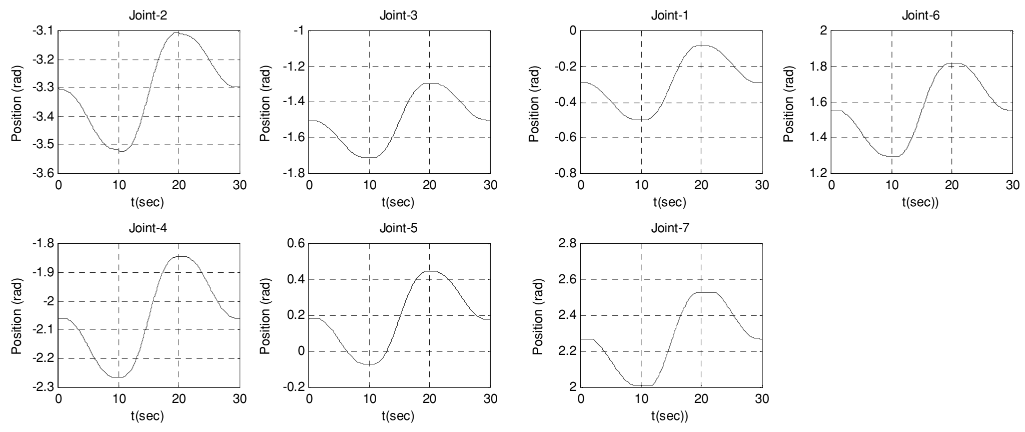


Fig. 6.2. Joint positions for the self-calibration task.

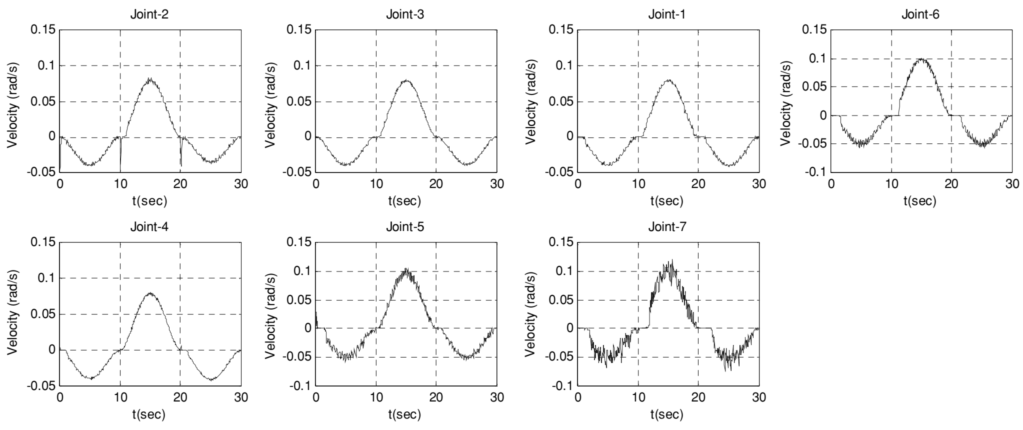


Fig. 6.3. Joint velocities for the self-calibration task.

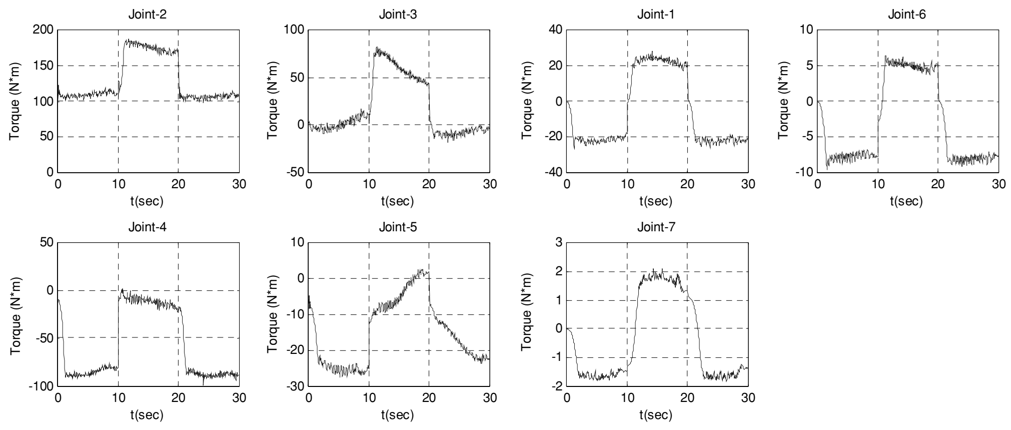


Fig. 6.4. Joint torque for the self-calibration task.

communication and process synchronization, is achieved very well. Visual observations indicate that the movement of the REDIESTRO manipulator is very smooth.

6.4. Task #2: peg insertion

A peg insertion task performed by REDIESTRO #2 was investigated. The goal was to demonstrate the feasibility and effectiveness of the design and implementation presented in the previous sections for an advanced application task. The second physical implementation using the SCRAM Network as Robot Group Shared Data was adopted as the communication medium for data exchange between the SSC and the ESC. The participating processes are the SSC Worker, the SSC Manager, the SSC DDM, the ESC Data Acquisition, the ESC Command Sender & Safety Guard, the ESC Manager, and the ESC DDM.

6.4.1. Task segments. There are in total four segments in this task: Approach, Insertion, Pullback and Home. Table 6.III lists the sub-task, control scheme and duration of each segment.

6.4.2. Frequencies. The SSC and ESC both run at 200 Hz. As described in the previous sections, the SSC and ESC are not synchronized with each other. The data-updating rate by the SCRAM Network through coaxial cables was set at 1000 Hz. However, there are no documented ways to determine when the update actually occurs. Therefore,

Table 6.III. Segments of the peg insertion task.

Segment	Subtask	Control scheme	Duration (Seconds)
1	Approach	Position control	20
2	Insertion	Force control	28
3	Pullback	Force control	12
4	Home	Position control (PID)	18

a higher frequency (greater than 200 Hz) of data updating by each DDM at the sub-controller node had to be used in order for the sub-controllers to get the updated data in time. As indicated by the observations in the experiment, the higher the frequency, the better was the system performance. The frequency at which the DDM updated the Local Data module was chosen to be 1000 Hz in this experiment.

6.4.3. Analysis of the results. The plots in Figures 6.5 through 6.14 show the experimental results. These consist of the positions, velocities, accelerations, and torque commands for the joints, the end-effector force and torque, and the end-effector position and orientation.

a. Effectiveness of position control. The goal position of the end-effector at the end of segment #1 ($t = 20$ seconds) in the three directions of the world frame was specified to be (0.215 m, −0.44 m, 0.16 m) and the orientation was specified to be

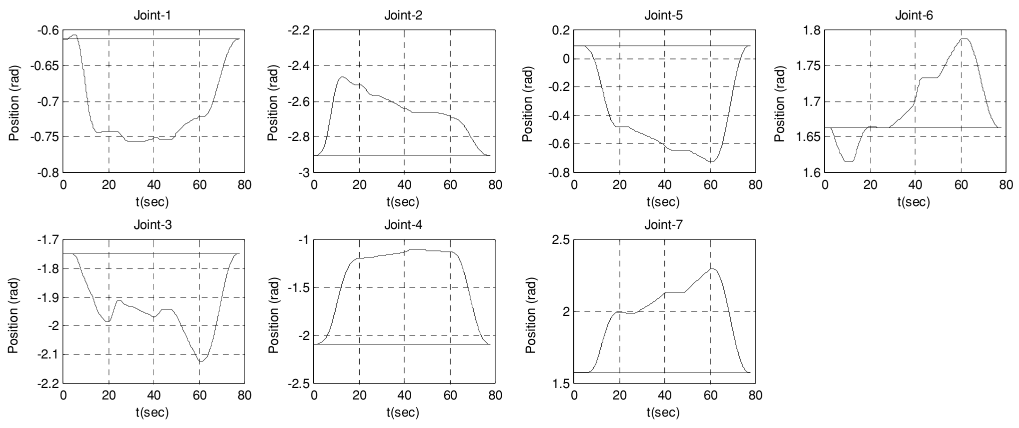


Fig. 6.5. Joint positions for the peg insertion task.

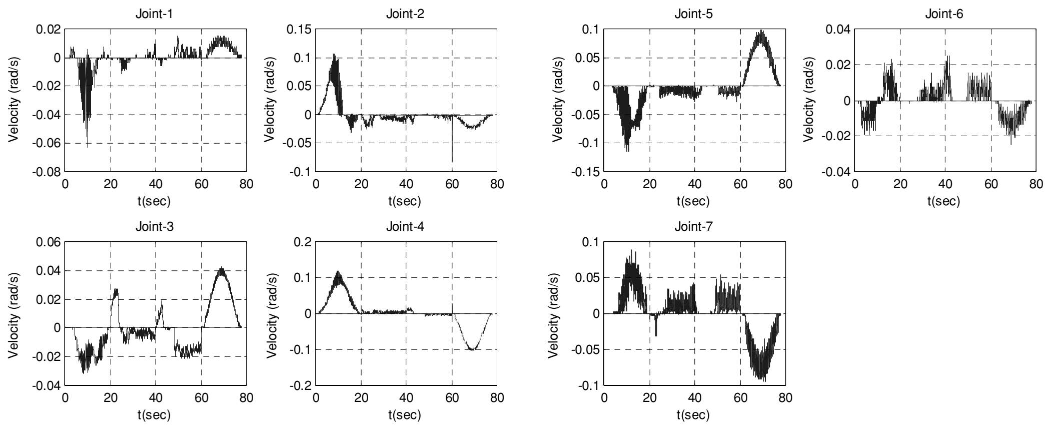


Fig. 6.6. Joint velocities for the peg insertion task.

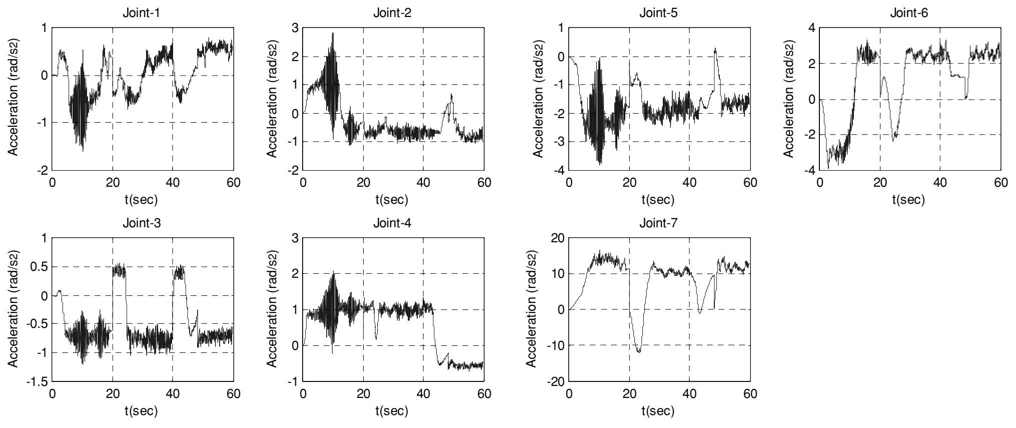


Fig. 6.7. Joint accelerations for the peg insertion task.

(−1.57 rad, 1.57 rad, 3.14 rad). As shown in Figures 6.13 and 6.14, at t = 20 seconds, the goal position was achieved with an error of about 0.0125 m in the X direction only. The error in achieving the goal pose was mainly the result of the kinematic error due to joint flexibility.

b. Effectiveness of force control. In the insertion segment (segment #2, from 20 to 48 seconds), the manipulator was in force control mode. The goal force in the three directions of the world frame was specified as (0, 0, −20) N and the desired torque as (0, 0, 0) N-m. In the period from 24 seconds to 40 seconds, the peg has landed on the edge of the hole but has not reached the center of the hole. It can be seen in Figures 6.11 and 6.12 (or Figures 6.9 and 6.10 for the tool frame) that the goal force has also been achieved. In addition, as shown in Figures 6.13 and 6.14, during the same period, while under impedance (force) control, the robot adjusts itself in its position and orientation in order to maintain the goal force. The adjustment leads the peg to reach the center of the hole and compler the insertion. It can also be seen that

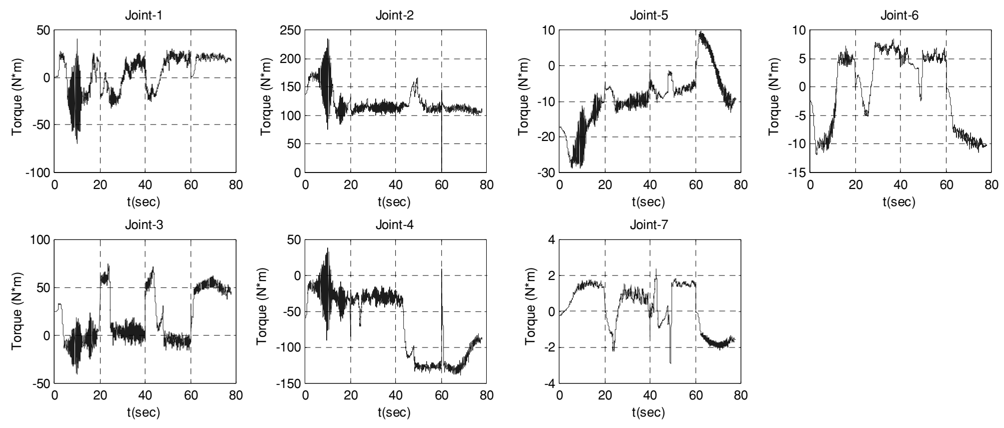


Fig. 6.8. Joint torque for the peg insertion task.

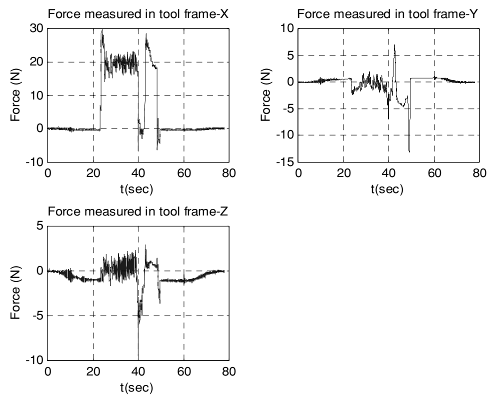


Fig. 6.9. End-effector force in tool frame.

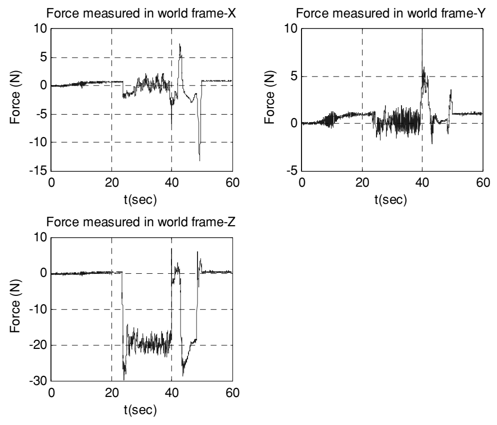


Fig. 6.11. End-effector force in world frame.

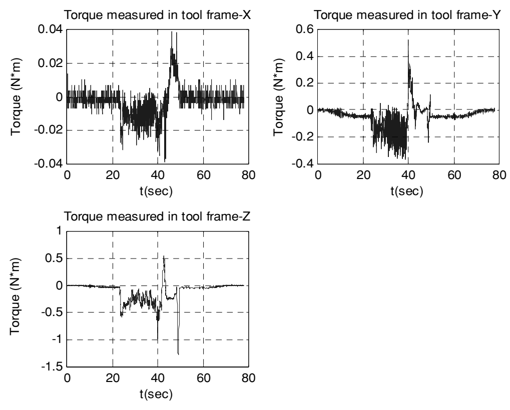


Fig. 6.10. End-effector torque in tool frame.

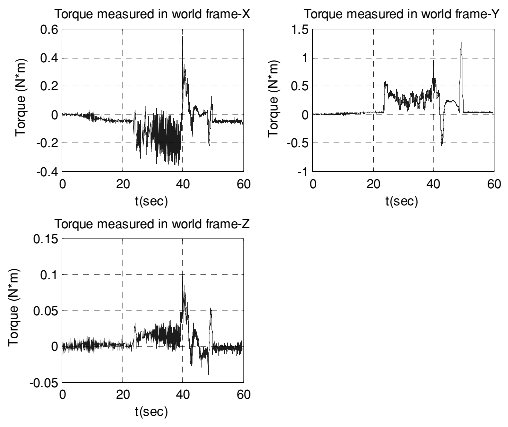


Fig. 6.12. End-effector torque in world frame.

after the peg has reached the bottom of the hole (between 45 seconds and 48 seconds), the goal force and torque are again achieved (Figures 6.11 and 6.12, or Figures 6.9 and 6.10).

c. Effectiveness of the real-time data exchange. The SSC and ESC run on the two computers separately. The data exchange between the two sub-controllers via the SCRAM Network has to be in real-time, that is, it must occur during every sampling and control period. The robot control commands (Figures 6.7 and 6.8) are computed by the SSC based on the robot feedback data and sent to the ESC to run the robot. The robot end-effector force feedback (Figures 6.9 and 6.10, or Figures 6.11 and 6.12) and the robot joint positions (Figure 6.5) and velocities (Figure 6.6) are acquired by the ESC and transferred to the SSC. The robot joint velocities are computed by digital differentiation. All the data are logged on the SSC computer node.

The effectiveness of the real-time communication (and control) can be seen from the plots of the trajectories, which the robot actually achieved (Figures 6.5, 6.6, and 6.7 through 6.14). For example, consider segment #4 (PID position control from 60 seconds to 78 seconds). The joint position trajectories are planned by the SSC using a fifth-order polynomial and the relevant joint command torque

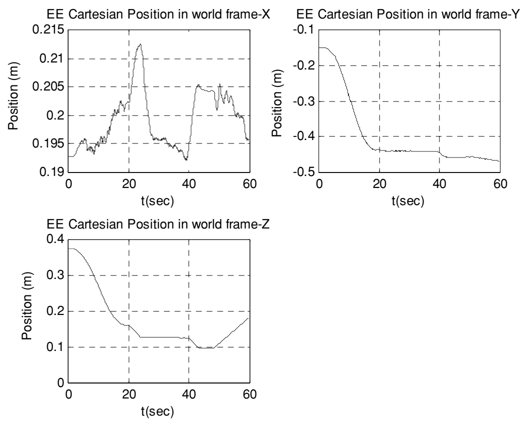


Fig. 6.13. End-effector position in world frame.

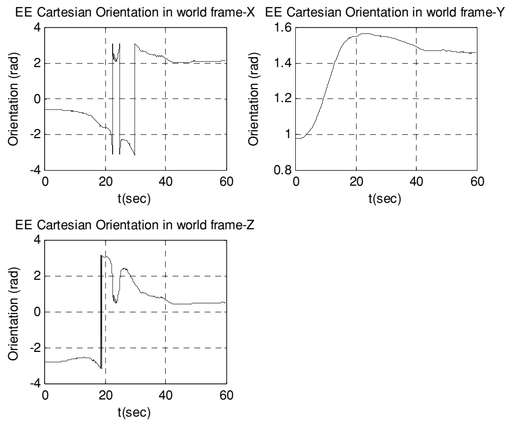


Fig. 6.14. End-effector orientation in world frame.

is computed and sent to the ESC for execution. From Figures 6.5 and 6.6, it can be seen that the actual robot kinematic trajectories coincide with the fifth-order polynomial very well. The effectiveness of the position and force control is actually based on the effective real-time data exchange.

d. Accuracy of the control. The experimental results indicate that the control scheme has good accuracy. The errors are within a reasonable range. There are kinematic errors in the REDIESTRO system because of the effect of joint flexibility, which mainly contributes to the position error of the goal position at the end of segment #1 ($t = 20$ seconds), as mentioned earlier. Another example is the way the robot joint velocities are acquired. The joint velocities are computed by digital differentiation; consequently, the quantization errors in the position sensing are amplified by the differentiation and the quantization errors of the delta time slots. There are also contributions to the inaccuracies of the control from the position and force sensors themselves.

6.5. Summary of the experiments

The self-calibration task was to test the implemented real-time multi-processing on a single computer node. The data exchange was achieved by the POSIX shared memory. The peg insertion task was to test the system with an advanced application task on the networked multiple computers. The data exchange is achieved by the POSIX shared memory and the SCRAM Network. The successful execution of the two tasks indicates that the multiple process cooperation and real-time information exchange, and the real-time synchronization are effective not only on a single computer but also on a computer network.

We have also successfully run other sophisticated tasks involving cooperative operation of the two REDIESTRO robots such as dual-arm peg-insertion, Velcro handling and collision avoidance.[4] The distributed real-time environment described in this paper has also been implemented at the Canadian Space Agency (CSA) in St. Hubert, Quebec, to perform cooperative control[18] of CSA's Automation and Robotics Testbed (CART), a highly sophisticated dual-arm system used for training and evaluation of advanced robotics methodologies. This and other applications strongly support the effectiveness and efficiency of the system architecture, design and implementation described in this paper.

7. CONCLUDING REMARKS

This paper is devoted to the software architecture design and system implementation of a reconfigurable system for real-time multiple robot control. Based on the analysis and modeling of a multi-robot control scheme, we partitioned the system into loosely coupled function units and data modules manipulated by the units. We have designed unified modular structures for the sub-controllers and controller processes, and constructed the system as a distributed real-time processing system. This architectural structure is aimed at facilitating an efficient implementation.

Under the framework of the architectural design and system implementation, all controller processes work autonomously. Intra-subcontroller information exchange is realized by a shared data module. All controller processes use the shared data module as a data repository. From the perspective of a controller process the data exchange with the outer world consists simply of reading data from and writing data to the repository. Inter-subcontroller information exchange is realized by the Dynamic Data Management (DDM) processes running on each subcontroller node. The DDM processes are independent of the subcontroller process group and are responsible for updating the data to the local shared data module. There is no need for explicit temporal synchronization among subcontrollers. The data dependencies are maintained by using datum-based synchronization. All controller processes are synchronized temporally, since the real-time nature of the control system requires synchronization for every sampling and control period. The data consistency is realized by using mutual exclusion. Moreover, the system is designed such that different hardware resources only relate to specific processes. Thus, all controller processes can be easily prioritized and scheduled independently, and the control system can be reconfigured to meet different functionality requirements. For instance, we can add new controller processes to the system to expand the functionality. We can also remove some processes without affecting the system integrity, provided the new functional requirements are satisfied. The controller processes can also run either on a single computer system or on a distributed computer network. These features ensure that the system meets the requirements of schedulability, predictability, scalability and adaptability. Effective schemes for software fault detection, fault anticipation and fault termination are included to meet safety and reliability requirements.

The implementation uses a QNX-based system on a PC architecture. We have used POSIX-compliant facilities whenever possible to improve the portability of the system. All the system components are modularly designed, and can be developed and tested independently. All these are intended to meet cost-effectiveness requirements.

The implemented control system has been tested and validated on an experimental redundant dual-arm robotic test-bed (the REDIESTRO manipulators) by performing a self-calibration task, a peg-insertion task and other complex dual-arm tasks requiring cooperative operation of the two redundant manipulators. The results obtained indicate that the implemented system is suitable for advanced real-time application tasks. Further validation of the effectiveness of the real-time environment has been obtained through its adoption and use by the Canadian Space Agency (CSA) for single- and dual-arm control[18] of CSA's Automation and Robotics Testbed (CART). These practical applications strongly support the claim that the system architecture, design and implementation described in this paper are effective, efficient, and reliable.

Acknowledgements

This research was performed at the University of Western Ontario and supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada under Grant CRD-172884 and by a contract with Bombardier Inc. under the Canadian Space Agency's Strategic Technologies in Automation and Robotics (STEAR-12) program. The contributions of F. Shadpey, H. Xie and I. J. Bryson in the development and initial implementation of the control algorithms for single- and dual-arm operation are gratefully acknowledged.

References

1. P. Fiorini, H. Seraji and M. Long, "A PC-based configuration control for dexterous 7-DOF arms", *IEEE Robotics and Automation Magazine* **5**, 30–38 (1997).
2. D. Lim and H. Seraji, "Configuration control of a mobile dexterous robot: Real-time implementation and experimentation", *Int. J. Robotics Research* **16**, 601–618 (1997).
3. D. Lim, T. Lee and H. Seraji, "A real-time control system for a mobile redundant 7-DOF arm", *IEEE International Conference on Robotics and Automation* (1994) pp. 1188–1193.
4. H. Xie, "Real-Time Cooperative Control of a Dual-arm Redundant Manipulator System", *Ph.D. Thesis* (The University of Western Ontario, 2000).
5. J. Fraile, C. Paredis, C. Wang and P. Khosla, "Agent-based planning and control of a multi-manipulator assembly system", *IEEE International Conference on Robotics & Automation* (1999) pp. 1219–1225.
6. J. Roberts, P. Corke, R. Kirkham, F. Pennerath and G. Winstanley, "A real-time software architecture for robotics and automation", *IEEE International Conference on Robotics & Automation* (1999) pp. 1158–1163.
7. A. Traub and R. Schraft, "An object-oriented real-time framework for distributed control system", *IEEE International Conference on Robotics & Automation* (1999) pp. 3115–3121.
8. R. Schraft, C. Schaeffer and T. May, "The concept of a system for assisting elderly or disabled persons in home environments", *24th IEEE IECON* (1998) Vol. 4. pp. 322–329.
9. F. Kolnick, *The QNX 4 Real-Time Operating System* (Basis Computer Systems, Canada, 2000).
10. R. Krten, *Getting Started with QNX 4 – A Guide for Real-Time Programmers* (QNX Software Systems, Kanata, Ontario, Canada, 1998).
11. B. Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns* (Addison-Wesley, Reading, MA 1999).
12. J. Mills, P. Baines, T. Chang, S. Chew, K. Lam and A. Rabadi, "Development of a robot control test platform", *IEEE Robotics and Automation Magazine* **2**, 21–28 (1995).
13. *QNX OS System Architecture* (QNX Software Systems Ltd., Kanata, Ontario, Canada, 1997).
14. *SCRAMNet+ Network* (Systran Corporation, http://www.systran.com/real-time.htm, 2000).
15. J. Angeles, F. Ranjbaran and R. V. Patel, "On the design of the kinematic structure of seven-axes redundant manipulators for maximum conditioning", *IEEE International Conference on Robotics and Automation* (1992) pp. 494-499.
16. F. Ranjbaran, J. Angeles, M. A. Gonzalez-Palacios and R. V. Patel, "The mechanical design of a seven-axes manipulator with kinematic isotropy", *Journal of Intelligent and Robotic Systems* **14**, 21–41 (1995).
17. M. Arenson, N. Arenson and J. Angeles, "Design and manufacturing of REDIESTRO 2, a seven-axis manipulator", *Technical Report TR-CIM-97-07* (McGill University, 1997).
18. Bombardier Aerospace Inc., "STEAR-12: Enhancements to the MSS Ground Support Facilities", *Final Technical Report for Contract No. 9F028-8-4900/002/XSD*, Space Station Strategic Technologies in Automation and Robotics (STEAR) Program (2000).