2

CONF-930403--35

A Portable Modular Architecture for Robotic Manipulator Control*

CONF-930403--35

DE93 010819

Philip L. Butler
Oak Ridge National Laboratory†
Robotics & Process Systems Division
P.O. Box 2008, Bldg. 7601
Oak Ridge, Tennessee 37831-6304
Telephone 615-574-4666
Facsimile 615-576-2081

To be presented at the
American Nuclear Society
Fifth Topical Meeting On
Robotics and Remote Systems
Knoxville, TN
April 26–29, 1993

*Research sponsored by the Robotics for Advanced Reactors Program of the Office of Nuclear Energy, for the U.S. Department of Energy.

†Managed by Martin Marietta Energy Systems, Inc., for the U.S. Department of Energy under contract DE-AC05-84OR21400.

A PORTABLE MODULAR ARCHITECTURE FOR ROBOTIC MANIPULATOR CONTROL*

Philip L. Butler
Robotics & Process Systems Division
Oak Ridge National Laboratory†
P.O. Box 2008, Bldg. 7601
Oak Ridge, TN 37831-6304
Telephone 615-574-4666
Facsimile 615-576-2081

ABSTRACT A control architecture has been developed to provide a framework for robotic manipulator control. This architecture, called the Modular Integrated Control Architecture (MICA), has been successfully applied to two different manipulator systems. MICA is a portable system in two respects. First, it can be used for the control of different types of manipulator systems. Second, the MICA code is portable across several operating environments. This portability allows the sharing of common control code among various systems. A major portion of MICA is the precise control of multiple processors that have to be coordinated to control a manipulator system. By having MICA control the processor synchronization, the system developer can concentrate on the specific aspects of a new manipulator system. MICA also provides standard functions for trajectory generation that can be used for most manipulators. Custom trajectory generators can be easily added to suit the needs of a particular robotic control system. Another facility that MICA provides is a simulation of the manipulator, allowing the control code to be simulated before trying it on a manipulator system. Using this technique, one can develop code for a manipulator system without risking damage to the arm during development.

INTRODUCTION

The Modular Integrated Control Architecture (MICA) (Butler 1992) is a framework in which real-time systems can be created. MICA makes the distinction between asynchronous and synchronous processes by providing appropriate control methodologies for each. Asynchronous processes are those for which exact timing synchronization among processes on one or more processor is not required. By taking advantage of this consideration, the implementation requirements can be reduced significantly over those for synchronous real-time processes. Synchronous real-time processes are those for which exact timing synchronization between processes on one or more processors is required for proper system operation. This constraint forces certain design considerations that are discussed below. Typically, a synchronous technique is required for robotic actuator control. Conversely, asynchronous techniques can be used for those processes that do not have to be time-synchronized.

Because synchronous processes are harder to design than asynchronous processes, it is desirable to use synchronous design techniques only when necessary. The interfaces among asynchronous processes are only functional interfaces, while the interfaces for synchronous processes must also include time synchronization as a factor.

For the asynchronous aspect of a real-time system, MICA provides an event mechanism that allows control information to be passed between any two processes in a real-time system. Those processes can exist on the same processor or rack, but they can transparently exist across a local-area-network (LAN) or a serial communications channel.

The motivation behind the asynchronous portion of MICA is based on the event-handling mechanism found in many graphical user interfaces, such as the Apple Macintosh (Apple Computer 1985). The Macintosh event system was used as a basis for the MICA event system. However, the MICA event system goes further in that it has been developed for distributed and multitasking systems. The Macintosh event system deals with such occurrences such as key clicks and mouse presses from the operating system while the MICA event system deals with control messages among a number of distributed processes.

Features of the asynchronous event system of MICA include the transparent communications of events from one process to any other process. If they don't deal with hardware, entire processes can be moved from one computer system to another for development and execution convenience simply by recompiling. A simple

*Research sponsored by the Robotics for Advanced Reactors Program of the Office of Nuclear Energy, for the U.S. Department of Energy.

†Managed by Martin Marietta Energy Systems, Inc., for the U.S. Department of Energy under contract DE-AC05-84OR21400.

event trace system can trace events and process responses for a large system, thus allowing for quicker integration capabilities.

The requirements for synchronous portions of a real-time system, of the type commonly used for low-level robotic and telerobotic control, are vastly different from those of the asynchronous portions of a real-time system. In these cases, MICA provides a framework in which synchronous functions can be properly sequenced among one or more processors in the same backplane.

The synchronous portion of MICA is based on the National Aeronautics and Space Administration (NASA) Laboratory Telerobotic Manipulator (Rowe et al. 1991), developed at Oak Ridge National Laboratory (ORNL). In this system, a framework exists for the precise control of synchronous functions required for the low-level control loops for robotic and telerobotic operation. MICA extends this framework by providing more functionality in the sequencing of operations at a synchronous level. Another motivation behind the synchronous portion of MICA is to allow for easier modifications and additions to the low-level control algorithms on a robotic system. Therefore, MICA encourages modularity at the synchronous level. In this way, for example, a researcher can actually apply a new inverse kinematics routine without affecting the remainder of the system.

Because of the modular design, new modules can be defined quickly as system development needs change. One big advantage of MICA is the capability to operate all of the synchronous code, except for the actual input/output (I/O) code, in a simulation mode. This greatly reduces the risks associated with operating a mechanism with previously untested code. Because MICA is extremely portable, another benefit is to allow development on a stand-alone personal computer before attempting to operate on the actual system. This mode uses the MICA simulation capabilities. The simulation capabilities can be used by a supervisory level graphical preview controller to simulate more accurately the actual path planning done by the robot system.

HISTORY

MICA was first developed for the HERMIES-III (Reister et al. 1991) system at ORNL in 1990. The asynchronous event system is used for the coordination of many processes among several computer systems, such as user-interfaces, wheel controllers, navigation, vision systems, sonar systems, etc. By having a single consistent control mechanism among these many processes, system development and debugging were greatly enhanced. The MICA event system utilizes the HELIX (Jones et al. 1992) heterogeneous distributed shared-memory system, also developed at ORNL, for communication across LAN. MICA is designed to be portable among operating systems. For example, the robot operates under the OS-9 operating system, while the user interface operated on a UNIX workstation.

The CESARm (Babcock et al. 1987) 7-degree-of-freedom manipulator, mounted on the HERMIES-III vehicle was the first system to use the synchronous capabilities of MICA. As a development aid, both in the initial development of the synchronous MICA framework and the porting of the existing nonmodular CESARm algorithms, a simple simulation capability was provided. Using this simulation capability, over 95% of the CESARm code was developed and debugged without having to test it on the actual manipulator. Only the actual I/O routines were not tested with this simulation capability.

In 1991, MICA was ported to VxWorks for use on the Environmental Restoration and Waste Management (ER&WM) Underground Storage Tanks-West (UST-W) 1991 demonstration controlling the large SPAR long-reach manipulator (Burks et al. 1991). In this system, MICA was used simply to translate high-level commands into the appropriate joystick commands that operated the SPAR manipulator. The SPAR subsystem was part of an integrated system based on the ER&WM Generic Intelligent System Controller (GISC). The asynchronous event system was used in a very limited way to interface with a local control computer. One of the benefits of this port of MICA was to show that it is easily ported from one operating environment to another.

ARCHITECTURAL REQUIREMENTS

Several design requirements are placed on the implementation of MICA. The first is that of modularity. Systems built with MICA must be modular so that systems can be expanded as requirements grow. Also, a modular design allows system designers to simultaneously develop and test code for unrelated segments of the system. For synchronous portions of MICA systems, the modularity must allow the safety subsystems to be separated from the control algorithms to allow several researchers to develop new control algorithms without having to duplicate the critical safety code.

The second requirement is that of portability. MICA concepts and code can be ported to different systems as system needs change and for different types of systems. By having this portability, the system hardware and operating environment can be changed with minimal impact on the large amount of robotic control code. By using the same MICA concepts among several systems, even though they may operate in different operating environments, significant portions of control code can be shared.

Integration of synchronous and asynchronous subsystems is another requirement placed on the MICA design. This requirement allows a simple asynchronous interface to a manipulator and thus allows complex motions to be programmed easily. By having an asynchronous interface to a synchronous subsystem, the high-level supervisory code can still treat such a subsystem in a similar manner as a pure asynchronous subsystem.

A major design requirement placed on MICA is the ability to simulate the operation of a synchronous robotic subsystem to verify proper code timing and validity before attempting to operate the actual hardware. This simulation capability allows researchers the ability to develop control algorithms off-line without having to be concerned about safety constraints.

ASYNCHRONOUS DESIGN

As mentioned previously, the asynchronous portion of MICA is built on the concept of an event. In simple terms, an event is a message from one process to another. This message can be considered to contain control information or data. Because data can be more efficiently communicated by using the existing HELIX system on HERMIES-III, an event is best suited to contain control information. The event record contains several pieces of information as shown in Fig. 1. The event code is the specific message for a particular process. Some event codes such as NULL_EVENT and QUIT_EVENT are common to all processes. Other event codes are specific to a particular process. For example, an event code may exist to command the sonar process to start a sonar scan. An event source and destination are included in the event record. This information is useful for an event-tracing process that monitors all events communicated in a distributed system. Also, the receiving process can check the source and decide whether the event is valid. Events are ranked so that high-priority events such as emergency shutdown supersede any lower priority events. Priorities also can be used to set the sequence of operations for a complex set of event messages. Also included in the event record is a data field for optional use. Processes can use this data field for anything necessary such as desired arm positions and sonar enable masks. One disadvantage of the data field is that if the event is to be transmitted to another type of system, consideration has to be given to the format of the data because floating-point formats and byte orders are different for various computer systems.

The sending and receiving processes must conform to the same message definitions. This is handled by using the same event definition header files across the entire system. Event types are defined to distinguish among the many processes in the system. Each event type, and therefore each process, has associated with it some specific event codes that it can respond to.

An event manager is used to control the delivery of events from one process to another. The event manager uses an array of priority queues—one for each event type—to hold events until they are received by the destination processes. The event manager receives an event posted by an individual process and places it into the appropriate priority queue, ranking as it is placed into the queue. When a process requests an event, the event manager pulls the top item out of that process's event queue and returns it to the process.

```
typedef unsigned short event_code;
typedef unsigned short event_type;
typedef unsigned short event_priority;

struct event_record
{
  event_code        code;
  event_type        source;    /* source process type */
  event_type        dest;      /* destination process type */
  event_priority    priority;  /* event priority */
  unsigned long     extra[14]; /* related event data */
};
```

Fig. 1. "C" Language event structure.

A small set of C language functions is used by the processes to use the event system as shown in Fig. 2. The function "register_event" is used by a process to identify itself to the event manager. If the event manager handles an event for an event type that has not registered, it assumes that the destination process is executing on another system. Therefore, the event manager broadcasts the event message to all other systems on the network. They receive the event and act on it accordingly. The "resign_event" function is used to notify the event manager that a process is about to quit. A process may need to determine whether an event is pending without actually pulling it off the event manager's priority queue. The "event_available" function returns a true flag if an event is available to be received. The process also can pass the address of an empty event record to get a copy of the pending event and can then determine whether it should process the event or defer it until a later time. The function "get_next_event" is used to actually receive an event. This function returns an event code that can be acted upon. Also, the process can pass the address of an empty event record to be filled in with the received event record including the optional data field. The process also can specify whether it expects to continue processing if an event is not available. Certain processes need only to act when they receive events. These are called foreground processes. Other processes, however, need to continue processing if an event is not pending. These are called background processes. If a background process has no event waiting for it, it receives a NULL_EVENT from the event manager. The reason for making the distinction between foreground and background is to allow the event system to more efficiently process foreground events. A process that is operating in the foreground can be put to

sleep until an event comes in for it. The last function needed for processes to use the event system is "post_event." Using this function, a process can send an event to any process, including itself. If an event does not use the optional data field, it passes a null pointer as the address of the event record. In this case, the "post_event" function will then create a temporary event record to contain the required information such as the event code, source, and destination.

```
void   register_event(event_type type);

void   resign_event(void);

char   event_available(struct event_record * dest);

event_code get_next_event(struct event_record * dest,
    short background_flag);

void   post_event(event_code event, event_type to_type,
    event_priority priority, struct event_record * src);
```

Fig. 2. Event function prototypes.

One benefit of the event-driven system is that of more consistent process implementation. Event processes are written in the form of an event loop as shown in the example of Fig. 3. The event loop receives an event from the event manager and then acts on it. This modeless design allows any event to be processed properly whenever it comes in. Note that in the example, the program uses the BACKGROUND mode of the "get_next_event" call only when the sonar is being scanned.

```
struct     event_record          my_event;

main()
{
  int      done = FALSE, block = BACKGROUND
  int      scan_flag = FALSE;
  event_code          event;

    register_event(SONAR);
    while (!done)
    {
          event = get_next_event(&my_event, block);
          switch(event)
          {
             /**** Common events ****/
             case NULL_EVENT:
                   break;
             case QUIT_EVENT:
                   done = TRUE;
                   break;
             /**** Process specific events ****/
             case SONAR_SCAN_ON:
                   block = BACKGROUND;
                   scan_flag = TRUE;
                   break;
             case SONAR_SCAN_OFF:
                   block = FOREGROUND;
                   scan_flag = FALSE;
                   break;
          }
          if (scan_flag)
              do_sonar_scan();
    }
    resign_event();
}
```

Fig. 3. Example event program

The event system of MICA is best suited for asynchronous control of processes on local and distributed systems. One constraint on its design was that it was to be portable to many different types of systems. Therefore, its implementation does not depend heavily on operating system services.

SYNCHRONOUS DESIGN

Much of MICA concentrates on the synchronous system design. MICA makes no constraints on the algorithms used for control, but rather provides a framework for their proper sequencing and execution. MICA uses function execution vectors to change the ordered list of functions that must be performed each time through a synchronous loop. By using these vectors, the entire operation of the synchronous subsystem can be changed simply and quickly. MICA decomposes a synchronous system into an organizational hierarchy, as shown in Fig. 4.

Cluster refers to an entire synchronous system. This distinction is necessary because an arm-control cluster may not necessarily be synchronized to a wheel-control cluster. The control code that must operate synchronously is contained within the cluster. Every cluster has an asynchronous interface to allow other processes to communicate with the cluster. This interface is able to interact with both the asynchronous system and the synchronous control loops within the cluster.

Modules are functional entities within a cluster. The collection of modules defines the cluster. A typical MICA system would have an I/O module, a SERVO module for motor-level servo loops, a KINEMATIC module for forward kinematic transformations, a FOLLOWER module for coordinate system transformations, a CONTROL module for trajectory planning, and a PLANNER module for proper sequencing of manipulator operations.

Packages are specific instances of modules. Installing different packages into a cluster typically changes the characteristics of the entire cluster. For example, a simulation package may be chosen for an I/O module instead of an actual drive package to perform simulations.

Routines perform the desired function of the package. One or more routines can exist in a single package. Several reasons exist for having multiple routines in a package. One is to allow a single package to have code executing over several processors. Another is to allow two different routines to execute in different orders on a single processor. For example, the actual I/O package may have a routine to do input. Another routine may then execute at a later time to perform safety checks and output to the motor drives. It is important to note that once the routines for a package are defined, a package can

then be treated as a single entity. Therefore, a package can be installed and removed without having to define explicitly which routines are affected.

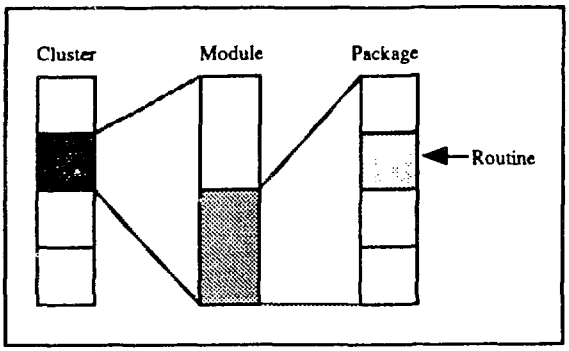

Fig. 4. Synchronous MICA hierarchy.

Figure 5 suggests the degree of modularity attained by MICA. This simple diagram, specific to real-time robotic manipulator control, shows the division between high-level control module, follower module, servo-level module, and I/O module. The output of the high-level control module is directed into the follower module, which performs appropriate transformations on the desired position data. The follower module feeds into the servo-level module, which might consist of a Proportional-Integral-Derivative controller. However, some connections may not be reasonable, because the "motor-level" follower is not being connected to the "torque controller" servo-level package. It is up to the system designers to decide which packages are compatible with each other. Figure 5 also shows that a simulation can be performed simply by choosing the simulation package instead of the actual I/O package. Additional modules may be added easily as needs warrant. For example, a real-time collision detection and avoidance module can easily be added without vastly affecting the remainder of the synchronous system.

Figure 6 shows a precisely timed sequence of operations as controlled by MICA. MICA ensures that the exact timing required to change the execution vectors (that control the type of operation) is carried out at the correct time. It is interesting to note that a high-level control process sets up this sequence by sending asynchronous events to the synchronous system. When the sequence is completed or an error occurs, the synchronous system sends an event back to the originating asynchronous process.

CONCLUSION

MICA establishes synchronous real-time control in tightly coupled multiprocessors and asynchronous real-time control on both local processors and processors connected over a LAN. The asynchronous events can be transmitted to remotely connected processors as well as to local processors.

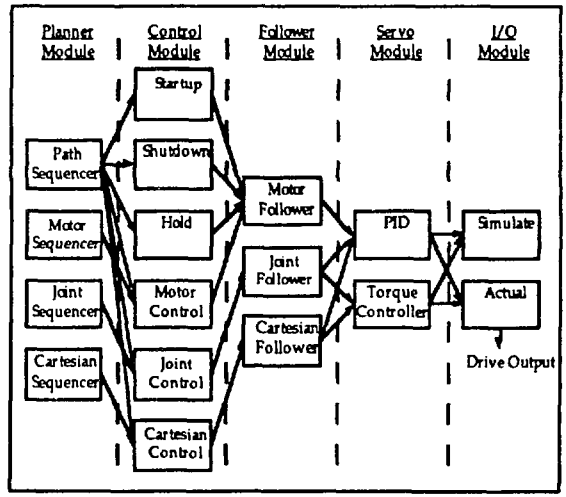

Fig. 5. Modular relationship between synchronous packages.

The development of MICA has successfully shown that complex robotic systems can be implemented with relative ease. MICA has proven itself in the successful HERMIES-III/CESARm implementation and the SPAR control interface to the integrated UST-W demonstration.

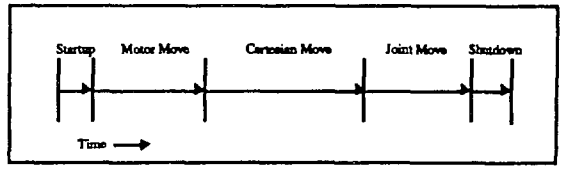

Fig. 6. Planner sequencing example.

All system requirements have been met in the design of both the asynchronous and the synchronous portions of MICA. The MICA modularity has been exploited in the design and integration of both asynchronous and synchronous subsystems. The portability of MICA concepts from one system to another encourages its use in future system designs.

A primary benefit of MICA is in system integration and maintenance. Large amounts of complex code are easily integrated into a single operational system. The integration philosophy of MICA has proven itself and will be exploited again. Because MICA encourages consistent coding techniques, large teams can effectively work together on final integration.

ACKNOWLEDGMENTS

I would like to thank John Jansen, Judd Jones, Reid Kress, John Rowe, David Thompson, and Michael Unseren for their helpful comments and Barry Burks, Bill Hamel, Reinhold Mann, and Frank Sweeney for their support and encouragement.

REFERENCES

Apple Computer, "Inside Macintosh, Volume I," Addison-Wesley Publishing Co., (1985).

S. M. BABCOCK, W. R. HAMEL, and S. M. KILLOUGH, "Advanced Manipulation for Autonomous Mobile Robots," ANS International Topical Meeting on Remote Systems and Robotics in Hostile Environments, 290–29 (1987).

B. L. BURKS, G. A. ARMSTRONG, P. L. BUTLER, and P. BOISSIERE, "Combined Long Reach and Dexterous Manipulation for Waste Storage Tank Applications," ANS 1991 Winter Meeting, San Francisco, Calif., (1991).

P. L. BUTLER, "An Integrated Architecture for Modular Control Systems," Robotics and Autonomous Systems, *10*, (2-3) 1992.

J. PP. JONES, A. BANGS, and P. L. BUTLER, "A System for Simulating Shared Memory in Heterogeneous Distributed-Memory Networks with Specialization for Robotics Applications," IEEE Conference on Robotics and Automation, Nice, France (1992).

D. B. REISTER et al., "Demo 89—The Initial Experiment with the HERMIES-III Robot," IEEE International Conference on Robotics and Automation, Sacramento, Calif. (1991).

J. C. ROWE, P. L. BUTLER, R. L. GLASSELL, and J. N. HERNDON, "The NASA Laboratory Telerobotic Manipulator Control System Architecture," ANS Fourth Topical Meeting on Robotics and Remote Systems, Albuquerque, N. M. (1991).