Presented at the 1994 IEEE International Conference on Robotics and Automation

# Standardizing I/O for Mechatronic Systems (SIOMS) using Real Time UNIX Device Drivers

Edward J. Nicolson
Department of EE&CS
U.C. Berkeley
Berkeley, CA 94720

## Abstract

*This paper describes a solution to controlling and communicating with an automated workcell based on a customized commercially available real time operating system. Two workcells, one for assembly and the other for machining, have been successfully controlled and integrated using much of the same software despite different applications, actuators, sensors, and computer hardware. Portability to different computer hardware arises from support by the operating system vendor while application portability arises from the development of this paper: standardized input and output with the mechatronic devices (SIOMS). The real time UNIX device drivers used to create the standard encapsulate complexities of I/O and present users with a simple interface which allows concurrent access to the devices from multiple processes. Recent advances in personal computer technology make this a cost effective solution.*

## Introduction

Suppose you are faced with the task of taking a collection of mechatronic components such as stepper motors, dc motors, binary and analog sensors and actuators and combining them in such a way as to automate an assembly or production sequence. The motion control algorithms are simple: acceleration, velocity, position, and force control on a per axis basis. It is important that the axes move synchronously. The usual systems integration solution to this problem is to connect the several different electronic interfaces of the various controller boxes to a computer. Each interface will be different and will require knowledge of special hardware interface commands. Due to the low bandwidth of some interfaces, such as serial, exact motion synchronization is difficult. Finally, you wish to perform analysis of the system efficiency, but it is not possible to access to the state of the actuators and sensors from a separate monitoring process.

It has been pointed out by many developers of real-time systems that system development time would be greatly reduced with encapsulation and standardization of input and output specific code [10, 13]. This paper proposes a possible standard interface through device drivers. The challenge is to create an interface standard that does not require an inordinate amount of overhead. The device drivers discussed in this paper allow efficient use of the processor with minimal overhead for interfacing to the devices at rates less than 100Hz.

The SIOMS driver solution was originally developed to address problems of maintenance, portability, concurrent access to sensors and effectors, control mode switches, and encapsulated input and output that existed in our previous workcell controller for a Sawyer motor small parts assembly system [2]. The previous controller was based on a modification of NYMPH [1], a 68020 VME Bus based multiprocessor system. Experience with that system indicated that for many of the tasks in manufacturing high performance, dynamic control is not as important as ease of integration of sensors and actuators to accomplish a task. Also we discovered that applications were not portable since they relied on specific hardware for I/O. We therefore decided to focus on providing the most portable solution at reasonable cost.

This paper proceeds first by discussing the operating system requirements of mechatronic systems and the applicability of a real-time UNIX. We follow with a description of the SIOMS driver protocol and their performance. We conclude by discussing applications and possible extensions.

## Mechatronics

Mechatronics is a field which originated in Japan as the study of the combination of mechanical and electronic systems [4, 6]. A mechatronic device is a transducer or actuator that has an electronic interface. Examples of mechatronic devices include stepper motors, DC motors, light beam sensors, force and torque sensors, tactile sensors, tactile stimulators, and pneumatic grippers. These simple components are used in larger devices such as mills, lathes, conveyor belts, and robotic manipulators.

We can classify the basic types of mechatronic devices as axis, binary, or analog. Examples are shown in Table 1.

| Mechatronic Device | Type |
|---|---|
| LED Beam Sensor | Binary |
| Force/Torque Sensor | Analog |
| Stepper Motor | Axis |
| DC Motor | Axis |

*Table 1: Mechatronic device types.*

The classification of the devices reflects the information the mechatronic device provides or controls not the type of communication required, be it serial, parallel, or shared memory. Following the object oriented paradigm, we seek an I/O solution that will encapsulate and hide complexities of communication and give devices a state and a set of methods to access them.

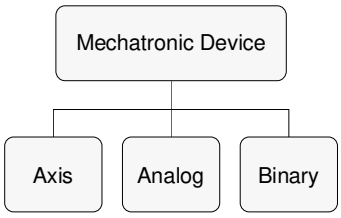


*Figure 1: Device class hierarchy.*

Conducting a robotic experiment or constructing a manufacturing workcell typically involves the computerized integration of several mechatronic devices. The integration requires the programming of time dependent tasks such as control loops and sensor event responses. The computational complexity of the tasks is low, however they must either occur at periodic intervals or respond with a minimum latency. This requires task scheduling on a multiprocessing operating system to be preemptive and prioritized. So as not to restrict communication with a device to one process at a time, the device I/O interface must allow concurrent access by multiple processes.

## Real time UNIX

Standard UNIX operating systems already provide the important features for portable open systems software development such as a graphical interface via X windows, networking via TCP/IP and NFS, and multiple processes and users. With the recent IEEE-1003.4 Posix real time extensions a standard full-featured development environment for real-time applications is available [12].

Besides offering a standard way of scheduling and communicating with real time tasks, a Posix 1003.4 compliant UNIX allows access to hardware via device drivers: pieces of code that are dynamically linked into the kernel of the operating system. The device driver mechanism is an ideal way to standardize I/O with mechatronic devices for the following reasons: 1) the I/O is functionally standardized through the use of UNIX system calls, 2) the operating system kernel provides robust features that automatically close open devices and return the I/O channel to an initial state when a user program that was accessing them exits or crashes unexpectedly, 3) I/O contention brought on by concurrent I/O requests by multiple processes is resolved with semaphores in the driver code, 4) more recent real-time UNIX operating systems allow system call interruption inside the kernel so that driver code becomes just an extension of the user code, without task-switching penalties, and 5) the drivers encapsulate and hide the special hardware knowledge required to communicate with mechatronic devices.

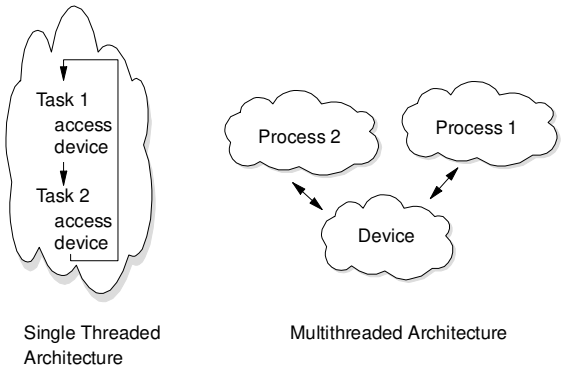


*Figure 2: Advantages of a multithreaded architecture. A single threaded architecture requires all tasks to be coded together. With multithreading and device drivers tasks may be added incrementally.*

It is the kernel and the standardized I/O through device drivers that provide much of the power of UNIX. Many of the tools that make up the executable utility programs of UNIX rely on device drivers for terminal, disk, and network I/O. By adding mechatronic device drivers and additional tools we simply extend the operating system's capabilities.

Once created for a particular piece of hardware a device driver may be reused in other applications, or used simultaneously by multiple applications. These applications programs, if general enough, may be taken to another system with drivers following the same interface allowing two forms of portability, one in the creation of input and output code, the other in the creation of programs that use the drivers.

## Using the SIOMS drivers

For the above reasons a set of device driver interfaces

specifically designed for mechatronic devices was created and termed "SIOMS drivers". Access to the devices is achieved with the 6 UNIX system calls: `open()`, `close()`, `read()`, `write()`, `ioctl()`, and `select()`. `open()` and `close()` perform the functions of initiating and finishing access to a device. `read()` and `write()` allow input and output respectively, `ioctl()` is used usually to modify the I/O and `select()` is usually used to block, or halt, the calling process until an event.

To get access to a device a process must first open a device node file. A device node is a special type of file that usually resides in the `/dev` directory of the file system. Suppose there is a motor driver, then the first motor node might typically have the device node file `"motor0"` and the C language code

```
motor0fd = open("/dev/motor0",O_RDONLY,0);
```

would assign the integer `motor0fd` to be the file descriptor to be used in all subsequent access to the motor. When the `open()` call is executed the UNIX kernel determines which driver the device node file corresponds to and executes the `open()` call registered for that particular driver. It is possible to have multiple device node files corresponding to the same physical device.

Besides associating a file descriptor with a set of driver functions, the open call also allows the driver to resolve device contention. The second argument of the open call is the mode which can either be `O_RDONLY`, `O_WRONLY`, or `O_RDWR` corresponding to read only, write only, and read and write access. To ensure that two processes do not try to control the same output, SIOMS output devices allows only one process to have write access to a device at a time. If a duplicate request occurs the system error code `errno` is set to `EBUSY` and `open()` returns -1.

The `close(fd)` call, where `fd` is the file descriptor to be closed, is used when the process is finished accessing the device. For the SIOMS axis driver this call will free up a particular axis to be opened by another process. As a safety feature the SIOMS axis driver ensures that all axes associated with an axis device node being closed are halted. This feature combined with the automatic closing of devices on process termination allows safe recovery from user program crashes.

The `read()` and `write()` calls are used in the following way:

```
read(fd, struct_size, struct_ptr);
write(fd, struct_size, struct_ptr);
```

where `struct_ptr` is a pointer to a structure used for access and `struct_size` is its size in bytes. The structures used for reading from and writing to the three different SIOMS driver types are listed in Table 2. The calls return the number of bytes read or written and -1 if there was an error. The most common error is an invalid structure field entry. Below we describe the specifics of reading and writing for each of the device types.

| | Reading | Writing |
|---|---|---|
| Binary | `struct binary_read{`<br>`  int struct_type;`<br>`  int data;`<br>`};` | `struct binary_write {`<br>`  int struct_type;`<br>`  int data;`<br>`};` |
| Analog | `struct analog_read{`<br>`  int struct_type;`<br>`  double data;`<br>`};` | `struct analog_write {`<br>`  int struct_type;`<br>`  double data;`<br>`};` |
| Axis | `struct axis_read {`<br>`  int struct_type;`<br>`  int axis;`<br>`  int in_motion;`<br>`  double position;`<br>`  double velocity;`<br>`  double k,kz,kp;`<br>`  double kb;`<br>`  double force;`<br>`};` | `struct axis_write {`<br>`  int struct_type;`<br>`  int axis;`<br>`  int control_mode;`<br>`  double position;`<br>`  double velocity;`<br>`  double acceleration;`<br>`  double jerk;`<br>`  double max_velocity;`<br>`  double max_acceleration;`<br>`  double time;`<br>`  double k, kz, kp, kb;`<br>`  double force_offset;`<br>`  double force;`<br>`  double force_velocity;`<br>`  double max_force;`<br>`};` |

*Table 2: Structures used to read from and write to the drivers.*

## Binary

The simplest type of driver is for binary sensors and effectors. A write to a binary device node file will set an output bit to the value specified by `data` in the `binary_write` structure. Similarly a read from a binary device node file will cause the state of an input bit to be written to the `data` field of the `binary_read` structure. Support for binary events is provided with the `ioctl()` call.

## Analog

Use of an analog device node file is functionally identical to a binary device node file. Output values are put in the `data` field of an `analog_write` structure and input values are read into `analog_read` structures. For the data to be correctly interpreted the system of units used in the driver must be known.

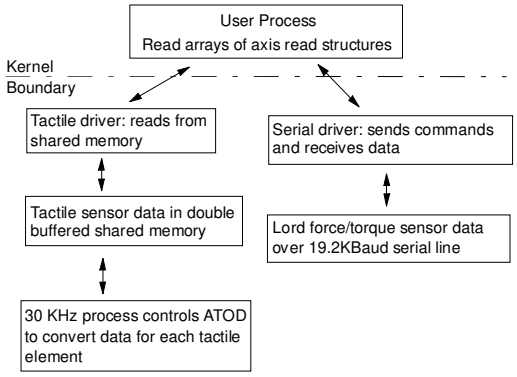


*Figure 3: Hiding handshaking details inside analog mechatronic device drivers.*

Though just as simple an interface as the binary device type, typically an analog device driver is more complicated internally. Consider for example two analog mechatronic devices, a tactile array sensor and a force/torque sensor. As shown in Figure 3, the tactile sensor data is collected by a 68020 processor and stored in shared memory on the VME bus whereas the force/torque data is retrieved over a serial line. The type of data, double precision numbers, is identical, yet the methods of retrieval is different. Fortunately the user is completely unaware of these complexities since they must only deal with the `analog_read` structures.

## Axis

The most complex of the drivers is that for an axis. We would like to communicate with DC servo, stepper and other motor controllers in the same way. Providing this rich set of features necessitates the complexity of the `axis_read` and `axis_write` structures shown in Table 2.

When a `read()` is executed on an axis device node file, the fields of the `axis_read` structure are filled in with the current position, velocity, controller gains, and force applied by the actuator. The `in_motion` flag indicates if the axis is currently moving in position or force mode or if it is being commanded in any other mode.

| Control Mode Flags | Description |
|---|---|
| `JERK, ACCELERATION, VELOCITY, POSITION, FORCE, or FORCE_VELOCITY` | Command the specified quantity for the axis. |
| `SPECIFIED_TIME` | The `time` field contains the duration for the command in seconds. If `time` < 0 then the command is executed until a new command is received. |
| `WAIT_FOR_EXTERNAL_SYNCH` | The command will be stored in the buffer but execution will be delayed until a synchronization signal is sent via an ioctl() call |

*Table 3: Some control mode options.*

The fields of the `axis_write` structure are self explanatory except for `control_mode`. The `control_mode` is set to the logical or of the flags shown in Table 3. Of course not all combinations of the flags are supported and the support will vary depending on the type of actuator.

Internally the axis driver is implemented as a high priority kernel thread and a command buffer. Writes to an axis device node file result in commands being appended to the buffer. If an array of structures is written to an axis device the commands will not be executed until the whole array is read into the buffer. In this manner complex trajectories may be specified. As the command blocks expire the controller moves onto the next command and handles any special requirements for switching between control modes. Figure 4 illustrates command buffering.

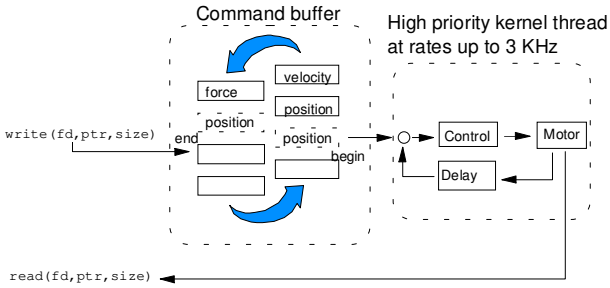


*Figure 4: Axis driver architecture*

Except in force mode, DC motors compute a desired position and velocity based on the current command. The motor torque is given by a PID law with velocity damping and torque offset:

$$\hat{\tau}[n] = k\big(e[n] + k_z e[n-1]\big) - k_p \tau[n-1] + k_b v[n] + \tau_o$$

$$\tau[n] = \begin{cases} \tau_m & \hat{\tau}[n] > \tau_m \\ \hat{\tau}[n] & -\tau_m < \hat{\tau} < \tau_m \\ -\tau_m & \hat{\tau}[n] < -\tau_m \end{cases}$$

where $e$ is the error between the actual and desired position, $v$ is filtered actual velocity, $\tau_o$ is the torque offset, $\tau_m$ is the maximum torque, and $\tau$ is the output torque. In force mode the output is simply the desired torque plus the velocity damping and offset torque. For stepper motors a desired position and velocity is computed and the output position slewed using a PID law on the error. The accelerations and velocities are thresholded according to the motor limits.

In addition to reading from and writing to an axis there are additional commands available through `ioctl()` for calibration, synchronization, and emergency stopping.

### Device type identification

Given the object based appearance of the drivers it is useful for them to be able to identify their type and number of nodes so utility programs can access them without being given this information by the user. For this reason a special `ioctl()` call is supported by all devices to return the device type. Table 4 gives examples of device node files currently in use.

| Device Node File | Type | Number | Description |
|---|---|---|---|
| `tactile0` | analog | 48 | tactile array sensor |
| `gripper0` | binary | 1 | pneumatic gripper |
| `mod0.readonly` | axis | 4 | read only node for a 4 axis manipulator |

*Table 4: Examples of device node files.*

## Driver performance

The important performance specifications to keep in mind when considering the SIOMS driver solution are operating system overhead and the number of axes that can be controlled simultaneously on one processor. These specifications will depend on the processor and bus used. To evaluate the system call overhead a simple program computed the average time to write a byte to the device `/dev/null`. Task switching overhead was computed by creating two processes and requesting them to yield alternately. The results for the 3 different processors used to date is shown in Table 5.

| Processor / Bus | Task switching | Write to device |
|---|---|---|
| 68030 / VME | 130 us | 86 us |
| 68040 / VME | 45 us | 20 us |
| 486DX266 / ISA | 18 us | 12 us |

*Table 5: Operating system overhead.*

For access to the drivers at rates less than 100Hz the 68040 and 486 provide insignificant overhead in task switching and system calls.

| | 68040, 25MHz VME | 486DX2, 66MHz, ISA |
|---|---|---|
| OS scheduling | 50us | 20us |
| Typical variable latency | 50us | 20us |
| Additional peak latency | 50us | 80us |
| Bus I/O, 12 axes | 70us | 26us |
| Control, 12 axes | 125us | 90us |

*Table 6: Breakdown of time from entry into the interrupt service routine until completion of the control update.*

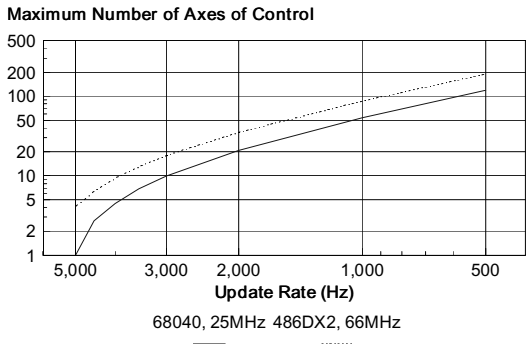


*Figure 5: Maximum number of axes of control as a function of update rate.*

To quantify the capability of the operating system to control multiple axes 4 events were timed in the control loop: 1) entry in to the interrupt service routine, 2) entry into the high priority controller thread, 3) completion of bus I/O, and 4) completion of control computation. The time from (1) to (2) is often termed the scheduling latency. Since scheduling latency depends on the total system load, it was monitored under two conditions, first while only the driver was running and second while a "tar xvf - / | rsh" process was running. Under the full system loading of the second case the scheduling latency increased by up to 100 microseconds. The results are shown in Table 6.

Based on these times one can estimate the maximum number of axes that can be controlled by a single processor at a given update rate, as is shown in Figure 5.

## Applications

SIOMS compatible drivers have been written for LynxOS [5] on a Motorola 68040 VME based system and an Intel 486 ISA bus system. The first set controls a small parts assembly workcell, the second controls a small machine tool and part handling system.

### Small parts assembly

The small parts assembly system consists of multiple 2 axis Sawyer (linear stepper) motors which move in the XY plane of a 0.8m x 1.2m platen. DC motors are attached to move in the z and $\theta_z$ directions. A total of 16 axes of motion are controlled at 1.8KHz. A variety of sensors and effectors, some of which are shown in Figure 6, are used for different robotic experiments. The following applications have all been programmed using different combinations of SIOMS drivers.

- Compliant control of screw threading: A force torque sensor and a module were used in a 25Hz servo loop to correct for positioning errors in screw insertion

[8].

- Localization and recognition of objects using cross beam sensing: Manipulator position was correlated with the breaking of 3 LED beams with 500us maximum latency [11].
- Derailleur assembly: SPARA, a LISP based world model controlled the assembly of a bicycle derailleur through TCP/IP.
- Peg-in hole: Using one 4 axis module to locate holes and another to handle pegs, multiple peg insertions were efficiently carried out. [9]
- Tele-taction experiments: A tactile sensor and a tactile display attached to different modules gave the sense of remote touch.
- Planar hand: A planar two fingered hand was controlled by grouping two 4 axis manipulators in synchronized motion.
- Pallet handling: Beam sensors on the conveyor belt alerted the modules to pick up arriving pallets.

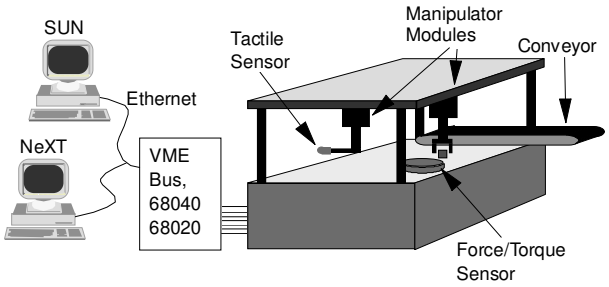


*Figure 6: Small parts assembly workcell.*

### Machining system

The second SIOMS system was created to replace serial line controllers for a machining workcell. A 486 PC controls the 24 axes of 3 mills, 3 lathes, a conveyor and a 6 axis arm. In addition limit switches and other binary sensors and effectors are used to monitor and control operations.

## Utility programs

The power of standardized drivers is not fully realized without a general set of tools to manipulate them. A set of utility programs have been written to provide a command line interface to the drivers. The `devin` program reads from a list of devices at a specified rate and prints the current state in ASCII format to the standard output. The `devout` program reads ASCII numbers from standard input and writes them to a list of devices. The `sdev` program gives a command line interface to the ioctl() options. In addition, synchronization points can be set in trajectories using the `smarkers` utility.

## Conclusions

The SIOMS driver specification has proved to be a useful and time saving development for programming of real time tasks in robotics and manufacturing. Its most significant contribution is standardization and encapsulation of the interface to mechatronic devices on an open, commercially available platform. It is clear that a driver specification like SIOMS could be added to the list of standardized I/O tools available on UNIX. Due to its axis based control it is not suitable for mechanisms with significantly coupled dynamics[7], however for simpler systems this is not a problem. The advantages are that upgrades to newer and faster processors are simplified. The solution is portable. It is quick to learn. Finally, since the system is open, drivers and applications may be freely written and shared.

## Acknowledgments

I would like to thank Richard Murray for his original NYMPH based controller, John Canny for the inspiration to create a new solution, and Steve Burgett, David Busing, Isabelle Mazon, Max Mendel, Dave Moore, Eric Paulos, and Aaron Wallack for comments on the driver specification. This work was supported in part by NSF Grants IRI-9114446 and IRI-9157051.

## References

[1] J.B. Chen, R.S. Fearing, B.S. Armstrong, and J.W. Burdick, "NYMPH: A Multiprocessor for Manipulation Applications," 1986 IEEE Conference on Robotics and Automation, April 1986, pp. 1731-1736.

[2] "Description of a Robot Workspace Based on a Linear Stepper Motor," AT&T Technical Journal, 67(2): 6-11.

[3] J.I. Egan and T.J. Teixeira, Writing a UNIX Device Driver, 2nd ed., John Wiley and Sons, 1992.

[4] C.J. Fraser, J.S. Milne and G.M. Logan, "An Educational Perspective on Applied Mechatronics," Mechatronics, 3(1):49-57, Feb. 1993.

[5] Lynx Real-Time Systems, Inc., Los Gatos, CA.

[6] D.K. Miu, Mechatronics: Electromechanics and Contromechanics, Springer-Verlag, 1993.

[7] R.M. Murray and S.S. Sastry, "Control Experiments in Planar Manipulation and Grasping," 1989 IEEE International Conference on Robotics and Automation, May 1989, pp 624-629.

[8] E. Nicolson and R. Fearing, "Compliant Control of

Threaded Fastener Insertion," 1993 IEEE International Conference on Robotics and Automation, May 1993, pp 484-490.

[9] E. Paulos and J. Canny, "Informed Peg-in-Hole Insertion using Optical Sensors," SPIE International Symposium on Optical Tools for Manufacturing and Advanced Automation, Sept. 1993.

[10] D.B. Stewart, D.E. Schmitz, and P.K. Khosla, "The Chimera II Real-Time Operating System for Advanced Sensor-Based Control Applications", IEEE Transactions on Systems, Man, and Cybernetics, 22(6):1282-1295, November 1992.

[11] A. Wallack, J. Canny, and D. Manocha, "Object Localization using Crossbeam Sensing," 1993 IEEE International Conference on Robotics and Automation, Atlanta, GA, pp. 692-699.

[12] T. Williams, "Real-Time Unix Develops Multiprocessing Muscle," Computer Design, 30(5):26+, March 1991.

[13] T. Williams, "BIOS standard would link real-time kernels to new hardware," Computer Design, 29(9):24+, May 1990.