To appear: *Proceedings of the 1998 SPIE International Symposium on Intelligent Systems and Advanced Manufacturing: Mechatronics Conference*, November 5, 1998, Boston.

Mechatronic objects for real-time control software development

Patrick F. Muir[a] and Jeremy W. Horner[b]

[a]Robotics Institute and the [b]Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA 15213

ABSTRACT

The design of real-time control software for a mechatronic system must be effectively integrated with the system hardware in order to achieve useful qualitative benefits beyond basic functionality. The sought-after benefits include: rapid development, flexibility, maintainability, extensibility, and reusability. In this work, we focus upon the interface between the device drivers and the control software with the aim to properly design this interface to best realize the aforementioned benefits. The results of this fundamental research include the development of an easily manageable set of four C++ object classes following an object-oriented approach to software design. These Universal Mechatronic Objects (UMOs) are applicable to a wide spectrum of actuators including dc motors, stepper motors, and solenoids; and sensors including pressure sensors, microswitches, and encoders. UMOs encapsulate the interface between the electrical subsystem and the control subsystem, providing the control software developer with a powerful abstraction that facilitates the development of hardware-independent control code and providing the electrical subsystem developer with an effective abstraction that facilitates the development of application-independent device drivers. Objects which are intuitively related to hardware components of the mechatronic system can be declared using the UMOs early in the system development process to facilitate the rapid concurrent development of both the electrical and the control subsystems.

Our UMOs were developed as part of a project to implement a real-time control system for a z-theta robotic manipulator. The z-theta manipulator is one component of the Minifactory project in the Microdynamic Systems Laboratory at Carnegie Mellon University. The goals of this agile assembly project include the reduction of factory setup and changeover times, plug-and-play type modularity, and the reuse of its components. The application of UMOs to the manipulator software development is shown to be consistent with these goals.

Keywords: object oriented programming, mechatronics, real-time control, mechatronic system design

1. INTRODUCTION

1.1 The growing role of mechatronics[1]

The development of systems incorporating mechanical, electrical, computer control and application software subsystems, is becoming commonplace. Today there are more solution options available to mechatronic system developers than ever before. Given a desired system functionality, there are often multiple solutions which will achieve the desired functionality using differing combinations of technologies. Instead of simply finding a satisfying solution, the opportunity to choose the "best" solution from among the alternatives is a reality. A solution which tightly integrates a set of highly interdependent components may be costly in terms of schedule and resources but result ultimately in the highest possible performance. On the other hand, a solution which is the straight-forward combination of existing technologies can minimize schedule and resource requirements, but does not take advantage of interdisciplinary synergies and results in suboptimal performance. We advocate a middle ground position in this paper, where interdisciplinary synergies are incorporated through proper configuration of subsystems, but the bulk of the development schedule and resources are devoted to realizing each subsystem from existing technologies. The intent being to achieve some desired synergies between subsystems while maintaining low resource and schedule requirements.

Specifically, we address mechatronic system development as a composition of subsystems which correspond roughly to the technical disciplines involved: mechanical, electrical, control and software engineering. Often a team approach is used, where each member is skilled in a relevant discipline. Given a functional specification for the system, and an understanding of the issues involved, compose the system from subsystems each of which can be reasonably achieved by the appropriate expert, then realize each of the subsystems and assemble the complete system. The synergy between subsystems of the system is built into the manner in which the system is composed from individual subsystems.

Once a synergistic set of subsystems has been configured, there remains the fundamental problem of specifying the interfaces between subsystems. As these integration and interfacing issues become more and more prevalent in research and development projects, it is apparent that new mechatronics tools and methodologies will be needed to deal with them. In this paper, we present one such tool: Universal Mechatronic Objects (UMOs). These UMOs can be applied to any mechatronic system to form the interface between the electrical and the computer control subsystems and thereby act as an effective foundation upon which the control and application software can be built.

In the following discussions, we will refer to the person/team responsible for implementing the electrical and controls subsystems respectively as the *electrical developer* and the *controls developer*. The discussions do not specify how many team members need to participate; in fact, a single person could act as both electrical developer and controls developer for some projects. Advantages accrue through the application of UMOs independent of the team's size and makeup.

1.2 The electrical subsystem/control subsystem interface

We are focusing upon the interface between the electrical subsystem and the control subsystem in this work. Figure 1 depicts the various hardware and software layers involved. The first software layer above the hardware consists of device drivers which interact directly with the computing hardware. The device drivers are considered part of the *electrical* subsystem because their development requires intimate knowledge of how the actuators and sensors are interfaced to the computing hardware. It is then efficient use of resources; therefore, that an electrical developer compose the device drivers. The control developer will need to read the sensors and write to the actuators, but need not be concerned with the details of how this reading and writing is accomplished.

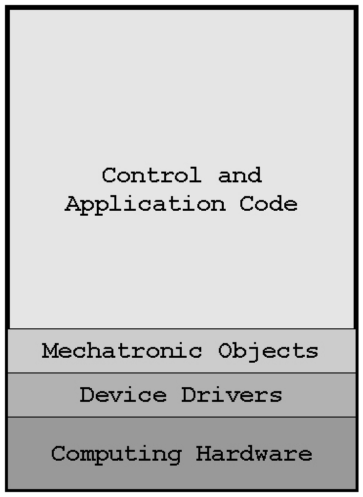

Figure 1: Layers of Software for a Mechatronic System

As can be seen in Figure 1, mechatronic objects are shown to be the interface between the device drivers (i.e., the electrical subsystem) and the control software (i.e., the control subsystem). Given this arrangement, we discuss in Section 4, how we designed the mechatronic objects to encapsulate the access functions for actuators and sensors so that the hardware-dependent details of the device drivers are hidden from view of the control developer. Further, the application-specific details of the control algorithms and application software are hidden from view of the electrical developer. The assembly of the mechatronic objects can be accomplished by the team member responsible for system integration, (i.e., a mechatronics engineer perhaps), early in the project, before either the electrical or control developers begin their work. Periodically throughout the course of the project the mechatronic objects can be re-evaluated based upon progress thus far. We propose that by defining the boundary between the electrical and control subsystems by the use of a standard set of UMOs, the entire project will proceed more quickly, efficiently and reliably toward the desired functional system realization.

1.3 Object oriented programming

We are advocating that an object oriented paradigm is appropriate for the definition of the electrical/control subsystem boundary. Object oriented programming, in contrast with more traditional structured programming methodologies is modular in a way which allows the assembly of all variables and functions relating to a concept. For example, an actuator (sensor) has associated with it certain variables and functions which all must be present for the programmer to use it effectively, but which are not necessary if the actuator (sensor) is not used. The required variables, called *member variables*, might include the actuator's name, the last value commanded to the actuator, and a variable whose value indicates whether the actuator is synchronous or asynchronous. The required functions, called *member functions*, might include an `initialize` function that sets-up the actuator for use, a `write` function, and a `finalize` function that is called when the program is about to exit. In such a case, it is useful to encapsulate these variables and functions into a single module, called an *object*. The C++ language has been designed to facilitate OOP with the built-in type *class*. An object that is derived from a particular class is called an *instance* of the class.

One advantage of the use of C++ and Object Oriented Programming (OOP) is the notion of inheritance. We can define a class which encapsulates all of the member variables and functions required for a wide spectrum of actuators and is entirely independent of any particular actuator. Then we can derive an instance of the class for each *particular* actuator in our system. Each of these actuator objects will *inherit* the member variables and functions of the parent class with no additional coding. The modularity enforced through the application of OOP allows software to be constructed so that improvements can be implemented through *local* modifications only. Thus for example, if changes are made to the device driver code, no changes will be necessary in the control code and visa versa. Also, because the programming objects (i.e, actuators and sensors) relate directly with the actual system components (i.e., actuators and sensors), no new abstract concepts need to be mastered. Further benefits will accrue if UMOs are applied consistently across multiple projects incorporating a wide variety of hardware because control software from one project can be readily reused on another.

1.4 Paper organization

We present our reasoning and implementation of UMOs as follows. Section 2 enters into a discussion of the motivations for our research in the area. Desired characteristics of both the system development process and the resulting system are introduced to motivate our work. Following this, Section 3 describes prior work which is relevant to the topic. Our four UMOs are detailed in Section 4. Here we provide complete printouts of C++ code which may be included and used in the software for any mechatronic system. Then in Section 5, we describe our application of UMO's to the development of a z-theta robot manipulator. This manipulator is an important component of an on-going project to develop a modular reconfigurable system for automated precision assembly called Minifactory. Finally, in Section 6, we conclude with a short discussion of the results.

2. MOTIVATION

2.1 Introduction

There are several possible approaches for interfacing the electrical and control subsystems. Consider this simple but prevalent situation: A mechatronic system incorprates a DC motor driven by an amplifier that is interfaced to the control computer by a Digital-to-Analog Converter (DAC) circuit. A control function is implemented to compute the voltage to next be applied to the motor. How do we write the software to command this voltage?

One solution would be to write an inline device driver within the control function that outputs an arbitrary 8-bit value to the DAC. Then, convert the voltage value into an appropriate 8-bit value and execute the device driver with the computed 8-bit value. This solution would function, but has many undesired characteristics. Firstly, the inline device driver code must be repeated in all of the functions which utilize the DAC which increases the size of the code unnecessarily. The control developer also needs to understand the operation of the DAC so as to convert his desired voltage into an appropriate 8-bit value. The more technology each team member must master, the greater are the chances for errors. It would be advantageous for the control developer to be able to write his code without needing to understand parts of the electrical subsystem. If ever the device driver needed to be changed, (for example, if a bug is found in the driver), then all copies of the device driver need to be located and changed. Such practices are prone to errors resulting from inconsistent coding of the device drivers.

An improvement on the aforementioned solution would be to write the device driver as a macro and allow the compiler to insert inline code everywhere the macro is invoked from the single master definition. This method ensures that all applications of the function are consistent, but does nothing to relieve the control developer from mastering parts of the electrical subsystem.

An even better solution might be to encapsulate the device driver in a function. This single function can then be called at runtime by the control software and any other software that uses the DAC. The device driver function could require a parameter which specifies the desired motor voltage in volts so that the control developer need not understand how to convert his voltage into an appropriate 8-bit value for the DAC. This solution is a widely-practiced method of solving the problem, because venders of DAC cards often provide device drivers with their products (as do vendors of many other types of interface cards). However, this solution still requires the control developer to know that a DAC is utilized in the commanding of the voltage to the motor. This issue may not seem very important in the case of one DAC and one motor, but with typical systems, multiple actuators with multiple types of interface circuits (e.g., digital IO (DIO), quadrature counters, ADC, and DAC circuits) are present and the control engineer is required to match IO channels to actuators and sensors and understand which electrical components are involved with each command of a voltage to a motor. Further, there is often more than one way to command a particular interface circuit to produce the same result (e.g., writing bits, bytes, or words) and sometimes values must be written in associated registers in order to achieve the results desired (e.g., data direction registers, output enable bits, etc.) This solution simply requires the control developer to master too much of the electrical system which has presumably already has been mastered by the electrical developer. And what if some part of the electrical subsystem is reconfigured? The control code must then also be changed.

Now consider a solution arrived at by proper application of the object-oriented paradigm. The *DAC-Amplifier-DC Motor* combination is declared as an object. Even though this object is not detailed until Section 4, we can describe the effects of its use here. If the object is one of a *standard* set of UMOs, the control developer may already be familiar with the interface which it presents, so he can apply it with no additional knowledge of how the results will be achieved. The solution is simply for the control engineer to include a standard header file in the file containing the control software, and at the point where he needs to command the voltage to motor he calls the member function `motor.write(voltage)` where `motor` is the name of the particular motor and `voltage` is the desired motor voltage. Similarly the electrical developer may already be familiar with the UMOs and he knows that he must write a device driver segment that takes as input a voltage in volts and executes all the conversions and low-level bit twiddling required to get that voltage to appear at the motor. This scenario is indeed an improvement over the prior methods. The control developer doesn't have to understand the operation of the DAC. In fact, he need not know that a DAC and amplifier are used. Even if the DAC and amplifier are replaced half-way through the project by a DIO and an amplifier with a direct digital input, the control engineer need not be concerned. Furthermore, the control developer can start working on his control software before the circuits are built, before the motor arrives, even before the electrical developer who will be writing the device driver joins the team. Likewise, the electrical developer can work independently. Moreover, there will be no disagreements over how values are passed between subsystems because this is part of the definition of the UMOs and is available for inspection by all beforehand.

We do not wish to imply that the control subsystem can be written in all cases without accounting for the transfer functions of the actual electronics used. However, it has been our experience that in most practical cases the *form* of the control software can be designed independent of the drive hardware and only gains need be adjusted for use with differing drive electronics and interface circuits.

Our intention here is merely to introduce some of the important issues relating to the specification of the electrical/control interface. In Sections 4 through 6, we provide complete declarations of the UMOs and the reasoning behind them.

3. PRIOR WORK

The subject of our work can be succinctly described as the application of OOP to the software interface between the device drivers and the control software for mechatronic projects. In this narrowly defined area, we are not aware of any similar works. However, prior publications in the areas of object-oriented programming (OOP) and the C++ programming language, OOP applied to control system design, and OOP applied to robotics projects are closely related precursors to our work. Here we review a representative sampling of these works.

Object oriented programming is a topic of much discussion and progress in the last few years. The advantages of OOP over more traditional structured programming methods is by now well documented [2, 3]. Although object oriented design can be implemented using several different programming languages, C++ is the language of choice[4, 5] for most of the literature that we have reviewed because of its many built-in features which facilitate OOP. As will be seen in Section 4, the salient characteristics of actuators and sensors can be mapped directly to member variables and member functions of object classes in C++ by application of the object oriented paradigm.

For the realization of industrial control systems, Ericsson[6] advocates the application of the object-oriented paradigm to all phases of control system development including functional specification, design, and implementation. He illuminates the application of an object oriented formalism in translating verbal system specifications into object definitions at a high level of abstraction. His work does not address individual actuators and sensors.

Pereira[7] similarly applies the object oriented paradigm to the development phases of a real-time industrial automation application. His assignment of physical equipment to software objects encompasses the lowest levels including individual actuators and sensors. However, his work introduces specific objects as they are needed by the system at hand and does not develop universal classes which are useful for a spectrum of actuators and sensors as we do. A software layer between the system hardware and the application software is introduced to perform device driver functions but no special interface is identified between the device drivers and the control code.

There is a growing list of projects employing OOP technologies in robotics. One of the earliest works was the development of a Robot Independent Programming Environment[8] (RIPE) at Sandia National Laboratories. The RIPE incorporated a four level programming structure and a hierarchy of generic parent classes. Their device driver level included such complex devices as bar code readers and gantry robots and so can not be equated to the much simpler, lower-level device drivers referred to in our work. The object classes within the higher software levels are documented but not for the lower levels. No generic actuator or sensor classes were published. Further, the interface between the lowest-level device drivers and the control code is given no special significance.

OOP has been applied to production control systems[9]. In the referenced work, the objects are intelligent manufacturing objects each of which represent a complicated piece of production machinary, such as a CNC turning machine. The authors do not model individual actuator or sensors. Their work focuses upon the scheduling, controlling and monitoring of the manufacturing objects.

Our research differs from these previous works. First and foremost, we are unique in our identification of the device-driver/control software interface as a boundary between engineering disciplines. As such, the design of the interface becomes a fundamentally important *mechatronic* issue. We have gone several steps beyond this realization to arrive at a universal set of objects (i.e., the UMOs) which can be applied to any mechatronic project to provide a functional, reusable, intuitive interface layer of software that facilitates system development. Also, our application of the object oriented paradigm is, in general, at a lower level than is previously documented.

Our results are similar to these previous works insofar as the benefits which accrue through the application of OOP. Ericsson[6] sites the intuition, reusability, useful abstraction, and faster development benefits that accrue. Pereira[7] concludes that information encapsulation, robustness, and reusability are some of the advantages realized. Miller and Lennox[8] site the benefits: reusability, extensibility, reliability and portability. Gausemeier[9], et.al. list the advantages of the application of OOP as flexibility, ability to distribute and scale the system, portability, and reusability.

4. UNIVERSAL MECHATRONIC OBJECTS

4.1 Introduction

We detail the design and application of our set of four UMOs in this section. Section 4.2 delves into a discussion of the characteristics which are desired for the electrical/control software interface. We review the reasoning which led us to decide what level of modularity is appropriate for mechatronic objects in Section 4.3. The nomenclature used for referring to different types of mechatronic objects is explained in Section 4.4. Then in Section 4.5, the UMOs are presented including complete listings of actual C++ code.

4.2 Desired characteristics for an electrical/control interface

The discussions in Section 2 illuminate some of the characteristics of a good electrical/control subsystem interface. Here we list all of the characteristics that we have identified to be useful for this interface. The interface should be:

- **functional -** the code must function as an interface between the electrical and control software subsystems.
- **universal -** the code must apply to a broad spectrum of actuators and sensors, and their associated electronics.
- **modular -** the code corresponding to a conceptual entity must be easily manipulated as a single unit.
- **intuitive -** the code must be easy to understand and apply, providing an appropriate level of abstraction.
- **complete -** the code should unambiguously define all aspects of the interface.
- **concise -** the code should incorporate only those features that are typically required in most applications.
- **extensible -** the code should allow future addition of custom member variables or functions.
- **efficient -** a reasonable attempt should be made to avoid excessive computational overhead.
- **compact -** a reasonable attempt should be made to avoid excessive memory requirements.

It should be noted that reuseablity derives directly from the universal and modular characteristics of the code. Reusability is the practical benefit that accrues when the code embodies both universal and modular characteristics. Similarly, rapid development and maintainability are practical benefits that result from the modular, concise, and intuitive characteristics of the code.

4.3 What are the objects?

The applicability of OOP techniques to a particular problem can be ascertained by the amount of commonality between the concepts in the problem space. It is, therefore, useful to identify the concepts and the commonalities involved in the device-driver/control software interface. Figure 2 depicts the generic interconnection of actuators and sensors within a mechatronic system.

Actuators, such as motors, solenoids, speakers, relays, LEDs, and valves receive controlled power for their operation by driver electronics of some type. For a DC servo motor, this may be a large, complex motor amplifier. For an LED, the drive electronics may consist simply of a current-limiting resistor. In order for the computer software to control the actuator, an output circuit is used. For motor amplifiers that require an analog voltage command, the output circuit would be a DAC. If the actuator is an LED or a relay, the output circuit would be a DIO.

In an analogous fashion, sensors, such as switches, temperature, pressure, humidity, position, velocity, acceleration and force sensors require filter electronics and an input circuit. For example, a shaft encoder requires a quadrature decoder/counter in order to count the number of pulses generated by the encoder. In this case, the input circuit is a DIO channel consisting of 8 or more parallel bits. For a microswitch, the filter electronics may consist simply of a current limiting resistor, and the input circuit is a single DIO bit.

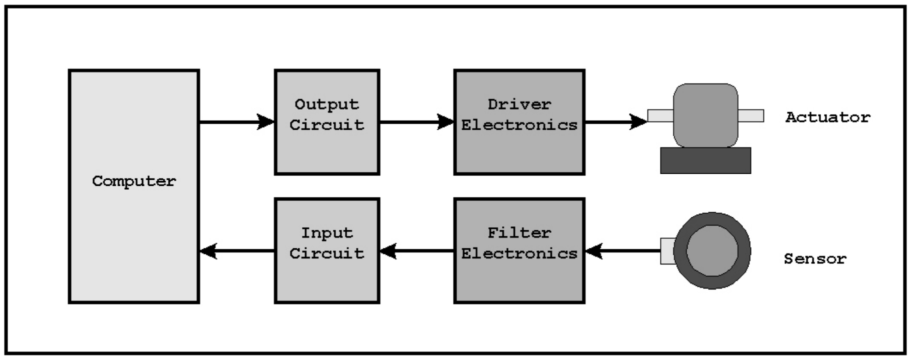

Figure 2: Interconnection of Actuators and Sensors within a Mechatronic System

The concepts that we select to be objects in our code should correspond to the concepts which have commonality that can be exploited to our benefit. Suppose we select a motor to be an object. The electrical interface to a motor consists of wires connected to its internal coils. A brushed motor has one set of two wires. A three-phase brushless motor may have 3, 4 or 6 input wires depending upon how it is internally wired. A stepper motor may have 4, 6 or 8 wires. We could similarly list differing control functions required by different motors. Clearly there is not much commonality evident in the selection of a motor as an object. The same is true if we were to select a specific sensor, such as an encoder, as an object.

Now consider grouping the drive electronics with the motor as an object. An amplifier for a brushless DC motor and one for a brushed DC motor may both accept DC analog voltages in the range -10V to 10V as inputs. Thus, we have more commonality at this level of abstraction than when assigning motors and encoders as objects. However, there are other difficulties at this level. Some DC servo motor amplifiers require digital inputs, some stepper motor drivers require a step bit and a direction bit, and others require a serial RS232 connection to command the motor. If we were to group individual sensors with their respective filter electronics, we would similarly find that some produce analog outputs in various configurations and others provide various configurations of digital outputs. We must conclude that this level of abstraction is also inadequate.

Consider grouping together the combination of a specific motor, drive electronics, and output circuit and call this an object. The interface to this object is now at the level of memory-mapped input/output circuitry. The command for an LED, solenoid, or valve is a single output bit. The command for a stepper motor is a set of bits. The command for a DC servo motor, brushed motor, AC motor or any other motor is a set of bits. By grouping a sensor with its filter electronics and input circuit we similarly find that in all cases the interface is a set of bits. This is a natural consequence of the fact that actuators and sensors are memory-mapped within the computer and thus accessed by reading and/or writing to memory locations. For this reason, there is a great deal of commonality at this level of abstraction. We have chosen to use this level of abstraction for our UMOs.

Listing 1: Common Type Definitions

```
//common.h
#ifndef COMMON_H
#define COMMON_H

typedef enum {ZERO, ONE, ON, OFF, RIGHT, LEFT, UP, DOWN, HIGH, LOW, FRONT, BACK,\
              CW, CCW, ENABLE, DISABLE, PASS, FAIL, COMPLETE, INCOMPLETE} binary_enum;
typedef enum {Z_ENCODER_COUNTER_INPUT, T_ENCODER_COUNTER_INPUT,\
              C_PRESSURE_SENSOR_INPUT, G_PRESSURE_SENSOR_INPUT, Z_MOTOR_DAC_OUTPUT,\
              T_MOTOR_STEP_OUTPUT, T_MOTOR_DIRECTION_OUTPUT, Z_ENCODER_ZERO_OUTPUT,\
              T_ENCODER_ZERO_OUTPUT, C_VACUUM_SOLENOID_OUTPUT, G_VACUUM_SOLENOID_OUTPUT,\
              G_PRESSURE_SOLENOID_OUTPUT, TIMER_INPUT} io_enum;
typedef enum {VOLTS, DEGREES, SAMPLING_PERIODS, MICRONS} units_enum;

#endif  COMMON_H
```

4.4 Nomenclature

The following naming conventions will be applied throughout the remainder of the paper to simplify our presentation. We will refer to the combination of a particular actuator with its associated drive electronics and output circuit simply as an “actuator”. Likewise, we will refer to the combination of a particular sensor with its associated filter electronics and input circuit simply as a “sensor”. Using this terminology, we must always be sure to identify all three components when describing an actuator or sensor. It is not complete to specify that an actuator is a DC motor; because a DC Motor which is driven by a PWM Amplifier through a single DIO bit requires a different interface than the same motor when it is driven by an analog amplifier through a DAC. We will use hyphenated notation as a shorthand method for referring to actuator and sensor configurations. For example, the two actuators just described can be denoted as *DIO-PWMamplifier-DCmotor* and *DAC-AnalogAmplifier-DCmotor* and a typical sensor encountered would be denoted as *ADC-LowPassFilter-Microphone*.

Listing 2: Universal Mechatronic Object Declarations: Actuators

```
//ioLib.h
#ifndef IOLIB_H
#define IOLIB_H
#include "common.h"
//-----------------------------------------------------------------------------------------
class binary_actuator
{
private:
  bool is_synchronous; //True if the actuator is synchronous, false if asynchronous
  binary_enum current_command; //If synchronous, the most recent value commanded
  binary_enum current_value; //Most recent value written to the actuator
  binary_enum default_value; //Value written to the actuator for reset
  binary_enum one_name; //The name for the 1 state, e.g. "on"
  binary_enum zero_name; //The name for the 0 state, e.g. "off"
public:
  io_enum name;
  binary_actuator(io_enum name_parameter, bool is_synchronous_parameter,\
                  binary_enum default_value_parameter, binary_enum one_name_parameter,\
                  binary_enum zero_name_parameter);
  ~binary_actuator();
  reset(); //Resets asynchronously the actuator
  bool write(binary_enum value);  //Commands a value to the actuator
  bool update(); //Writes the most recently commanded value to the actuator
  binary_enum read(); //Returns the value most recently commanded
}; //-------------------------------------------------------------------------------------
class digital_actuator
{
private:
  bool is_synchronous; //True if the actuator is synchronous, false if asynchronous
  float current_command; //If synchronous, the most recent value commanded
  float current_value; //Most recent value written to the actuator
  float default_value; //Value written to the actuator for reset
  float min_value; //Smallest value that can be commanded to the actuator
  float max_value; //Largest value that can be commanded to the actuator
  float resolution; //Smallest command increment
  float zero_offset; //Command producing zero output
public:
  io_enum name;
  units_enum units; //All values have these units
  digital_actuator(io_enum name_parameter, units_enum units_parameter,\
                   bool is_synchronous_parameter, float default_value_parameter,\
                   float min_value_parameter, float max_value_parameter,\
                   float resolution_parameter, float zero_offset_parameter);
  ~digital_actuator();
  reset(); //Resets asynchronously the actuator
  bool write(float value); //Commands a value to the actuator, returns true if ok
  bool update(); //Writes the most recently commanded value to the actuator
  float read(); //Returns the value most recently written to the actuator
};
```

In practice, software that is written to be easily understandable for accessing actuators which require only one output bit is significantly different than software that is intuitive for accessing those requiring a group of bits. The difference is that the binary state of the IO bit for actuators which use only one bit has a *logical* meaning whereas, for all other actuators the group of IO bits has a *numeric* meaning. For this reason, we refer to actuators which require a single output bit as *binary actuators* and those requiring a group of output bits are referred to as *digital actuators*. Similarly, Sensors requiring only one input bit are referred to as *binary sensors* and those requiring a group of input bits are *digital sensors*. In Section 4.5, it will become apparent why the software for a binary device is different from that for a digital device.

Our set of four UMOs (i.e., the binary actuator, digital actuator, binary sensor and digital sensor) is sufficient for modeling all devices which have a single interface value, for this reason we refer to this set of four mechatronic objects as the *atomic objects*. Some devices must be modeled as an combination of atomic objects. For example, a *DIO-StepperDrive-StepperMotor* actually requires two binary actuator objects, one binary actuator acts as the step signal and the second binary actuator acts as the direction signal. Another example would be a servo axis consisting of a *DAC-Amplifier-DCmotor* for axis drive and a *DIO-QuadratureCounter-Encoder* for axis feedback. We refer to mechatronic objects such as these which are formed as the combination of other mechatronic objects as *compound objects*.

Listing 3: Universal Mechatronic Object Declarations: Sensors

```
//ioLib.h continued

//------------------------------------------------------------------------------------------
class binary_sensor
{
private:
  bool is_synchronous; //True if the sensor is synchronous, false if asynchronous
  binary_enum current_value; //the most recent value read
  binary_enum one_name; //the name for the 1 state, e.g. "on"
  binary_enum zero_name; //the name for the 0 state, e.g. "off"
public:
  io_enum name;
  binary_sensor(io_enum name_parameter, bool is_synchronous_parameter,\
                binary_enum one_name_parameter, binary_enum zero_name_parameter);
  ~binary_sensor();
  binary_enum read(); //Reads a value from the sensor
  bool sample(); //Reads directly from the sensor and stores in current_value
  reset(); //Resets asynchronously the sensor
}; //---------------------------------------------------------------------------------------
class digital_sensor
{
private:
  bool is_synchronous; //True if the sensor is synchronous, false if asynchronous
  float current_value; //If synchronous, most recent value read
  float min_value; //Smallest value that can be read
  float max_value; //Largest value that can be read
  float resolution; //Smallest value change that can be read;
  float zero_offset; //Value read corresponding to zero
public:
  io_enum name;
  units_enum units; //All values have these same units
  digital_sensor(io_enum name_parameter, units_enum units_parameter,\
                 bool is_synchronous_parameter, float min_value_parameter,\
                 float max_value_parameter, float resolution_parameter,\
                 float zero_offset_parameter);
  ~digital_sensor();
  float read(); //Reads a value from the sensor
  bool sample(); //Reads directly from the sensor and stores in current_value
  reset(); //Resets asynchronously the sensor
}; //---------------------------------------------------------------------------------------
#endif  //IOLIB_H
```

4.5 Mechatronic Object Declarations

We now present and discuss the declaration of the mechatronic objects. Listing 1 is an excerpt from a header file named `common.h` because the type declarations therein are common to all of the other files. This listing contains the enumerated type `binary_enum` which contains all of the names which may be used to denote the two states of a binary actuator or binary sensor. The use of words, such as “on” and “off” to represent the two states of a binary actuator is far more intuitive than the use of the numbers 1 and 0. Similarly, the enumerated type `io_enum` facilitates the use of intuitive names for referring to actuators and sensors. We could have used the standard type `string` for naming them, but the computational overhead associated with the manipulation of strings is not warranted. The enumerated type `units_enum` enables the use of names for specifying the units for actuator and sensor values. This listing was taken from the code for the z-theta manipulator, so the actual enumerated entries may be different for a different mechatronic system. The use of these types will become obvious in the following code listings.

Listing 4: Code Structure for Device Drivers

```
//ioMan.cc
#include "ioMan.h"
#include "common.h"
//------------------------------------------------------------------------------------------
void IO_initialize(io_enum name_parameter)
{ switch(name_parameter)
  { case FIRST I/O_NAME: INITIALIZATION CODE FOR FIRST I/O; break;
     case SECOND I/O NAME: INITIALIZATION CODE FOR SECOND I/O; break;
     :
     case LAST I/O NAME: INITIALIZATION CODE FOR LAST I/O; break;
     default: cout << "No initialize function for that name." << endl;
};};
//------------------------------------------------------------------------------------------
void IO_finalize(io_enum name_parameter)
{ switch(name_parameter)
  { case FIRST I/O_NAME: FINALIZATION CODE FOR FIRST I/O; break;
     case SECOND I/O NAME: FINALIZATION CODE FOR SECOND I/O; break;
     :
     case LAST I/O NAME: FINALIZATION CODE FOR LAST I/O; break;
     default: cout << "No finalize function for that name." << endl;
}; };
//------------------------------------------------------------------------------------------
void IO_reset(io_enum name_parameter)
{ switch(name_parameter)
  { case FIRST I/O_NAME: RESET CODE FOR FIRST I/O; break;
     case SECOND I/O NAME: RESET CODE FOR SECOND I/O; break;
     :
     case LAST I/O NAME: RESET  CODE FOR LAST I/O; break;
     default: cout << "No reset function for that name." << endl;
}; };
//------------------------------------------------------------------------------------------
long IO_read(io_enum name_parameter)
{ switch(name_parameter)
  { case FIRST SENSOR NAME: READ CODE FOR FIRST SENSOR; return(VALUE);
     case SECOND SENSOR NAME: READ  CODE FOR SECOND SENSOR; return(VALUE);
     :
     case LAST SENSOR NAME: READ  CODE FOR LAST SENSOR; return(VALUE);
     default: cout << "No read function for that name." << endl; return(0);
}; };
//------------------------------------------------------------------------------------------
void IO_write(io_enum name_parameter,long value_parameter)
{ switch(name_parameter)
  { case FIRST ACTUATOR NAME: WRITE CODE FOR FIRST ACTUATOR; break;
     case SECOND ACTUATOR NAME: WRITE  CODE FOR SECOND ACTUATOR; break;
     :
     case LAST ACTUATOR NAME: WRITE  CODE FOR LAST ACTUATOR; break;
     default: cout << "No write function for that name." << endl; break;
}; };
```

Listing 2 displays the declaration of binary and digital actuators. Listing 3 displays declarations for binary and digital sensors. Taken together these two listings form the basis for all mechatronic objects.

Looking at the class `binary_actuator`, we find six *private* member variables. These member variables are private, meaning that they can not, and need not, be accessed by any of the device driver or control software. The Boolean variable `is_synchronous` is either true or false indicating whether or not the particular actuator should be commanded synchronously each sampling period, or should be commanded immediately whenever the `write` function is called. The member variable `current_value` stores the last value that was written using the `write` function. `One_name` and `zero_name` are the names of the states corresponding to 1 or 0 binary values. The `default_value` is the name of the state in which the actuator will be initialized or reset to. The remaining variable, `current_command`, is the value that was last commanded to the actuator hardware. Note that if the actuator is asynchronous, the `current_command` will be assigned the `current_value` as soon as the `write` function is called. Whereas, a synchronous actuator will not change its `current_command` until the next time `update` is called, even if `write` has been called one or more times.

The only public member variable is the actuator's `name`. The `name` is used within the device drivers to make the proper correspondences between device driver code segments and mechatronic objects. The member function having the same name as the class, `binary_actuator`, is called the *constructor* according to C++ terminology. This is the function that gets called when the user declares a new object of this class, which explains why one parameter for each of the member variables is passed to the function. Upon calling the constructor a new object is created and the member variables are assigned to be equal to the parameters passed. The actuator is also initialized when the constructor is called. The function `~binary_actuator` is the *destructor*. This function is called to free up the resources allocated to the object after it is no longer needed. The function `reset` simply calls the device driver code segment `io_reset` which typically sets the value of the actuator to its default value, but may be programmed by the electrical developer for other duties as well. The function `update` commands the `current_value` to the actuator hardware and thus only applys to synchronous actuators. The `write` function gives the actuator a new value and returns true if it was successful. The `read` function returns the value of `current_command`.

The private member variables for the digital actuator differ from those of the binary actuator by the lack of `one_name` and `zero_name` and the addition of `min_value`, `max_value`, `resolution`, and `zero_offset`. `Min_value` and `max_value` correspond respectively to the minimum and maximum values that can be commanded to the actuator. The variable `resolution` denotes the command change corresponding to a one bit change. The `zero_offset` is the value that must be commanded to the actuator to obtain a zero response. There is one additional public member variable called `units` whose meaning is self-evident. The ability to specify the units of the actuator commands allows the control developer the ability to program using the units which are best suited for each particular actuator. Whereas all values within a `binary_actuator` are of type `binary_enum` and are interpreted using the specified values of `one-name` and `zero-name`, all values within a `digital_actuator` are of type `float` and are interpreted according to the `units` member variable.

A binary sensor is declared similar to a binary actuator except that no `default_value` or `command_value` are needed. Sensors do not have a `write` function nor an `update` function. In contrast, a `sample` function is included which applies only to synchronous sensors. The `read` function only reads the state of the actual hardware if the sensor is declared as asynchronous. Otherwise a call to `read` will return the `current_value`; the `current_value` itself will not be set from the actual hardware until the next execution of the `sample` function. Digital sensors behave like binary sensors insofar as sampling and reading are concerned, and like digital actuators in the way that units are used.

It is worth mentioning that the four classes could be declared hierarchically with a *device* class having the member variables `is_synchronous` and `name` and the function `reset`, because these are common to all the classes. Derived from the parent *device* class could be *binary_device* and *digital_device* classes which respectively declare additional commonalities. Then at a third level, the four atomic objects could be declared with only the member variables and functions not already declared by their respective parent classes. We have chosen not to make the declarations in such a hierarchical fashion because it has been reported[10] that the use of inheritance in this way incurs additional processor overhead that degrades the real-time performance, and because it tends to make the classes less intuitive. However, the user, at his discretion could rewrite the classes hierarchically and still enjoy all of the other benefits described in this paper.

The declarations for the mechatronic objects should be included in the control developer's code. The electrical developer has a different interface to deal with. In order for the mechatronic objects to access the device drivers properly, the device drivers must adhere to the simple format shown in Listing 4. The electrical developer must program four device driver code segments for each atomic actuator or sensor. Three of the required code segments are `IO_initialize`, `IO_finalize`, and `IO_reset`. Additionally, for each actuator he must provide `IO_write` and for each sensor he must provide `IO_read`. Once written, these code segments are simply inserted into the appropriate case statements shown in Listing 4. The code segment `IO_initialize` prepares an atomic object for use. Conversely, `IO_finalize` performs any tasks which are required when the object is retired from use.

Early in the development of a mechatronic system the integrator must declare each of the actuators and sensors in the system as an object. Compound objects must be modeled as an combination of atomic objects. For example, a servo axis might consist of a *D/A-Amplifier-DCmotor* for axis drive and a *DIO-QuadratureCounter-Encoder* for axis feedback. A compound object is declared by *aggregating* its component objects as will be shown in Section 5. One way to understand the roles of atomic and compound objects is by analogy with C++ types. The C++ language provides built-in types which can be applied directly, such as `char`, `int` and `float`. The user then is able to aggregate multiple chars, ints or floats within a `struct` to model compound data structures. Analogously, the UMO provides four built-in atomic objects (binary_actuator, digital_actuator, binary_sensor and digital_sensor) and the user is able to aggregate atomic objects in order to model combinations of these.

Once they are declared, each actuator (sensor) will *inherit* all of the member variables and functions defined by the *UMO*. This provides a powerful standard interface for use by the control developer even before any device drivers are written. In other words, the UMO's provide an abstraction for the control developer which allows the computer control software to be written in a portable, hardware-independent manner. Further, the electrical developer can compose the device drivers and format them to interface with the UMO's in a straightforward standard way as well. Thereby, the UMO's provide an abstraction for the electrical developer which allows the device driver software to be written in an application-independent manner. Central to the advantages of UMOs is the ability of the electrical and control developers to proceed with the development of their respective subsystems independently and in parallel. We propose that in practice this will result in a reduction in the total time for system development when compared with the traditional practice of not clearly defining this interface at all.

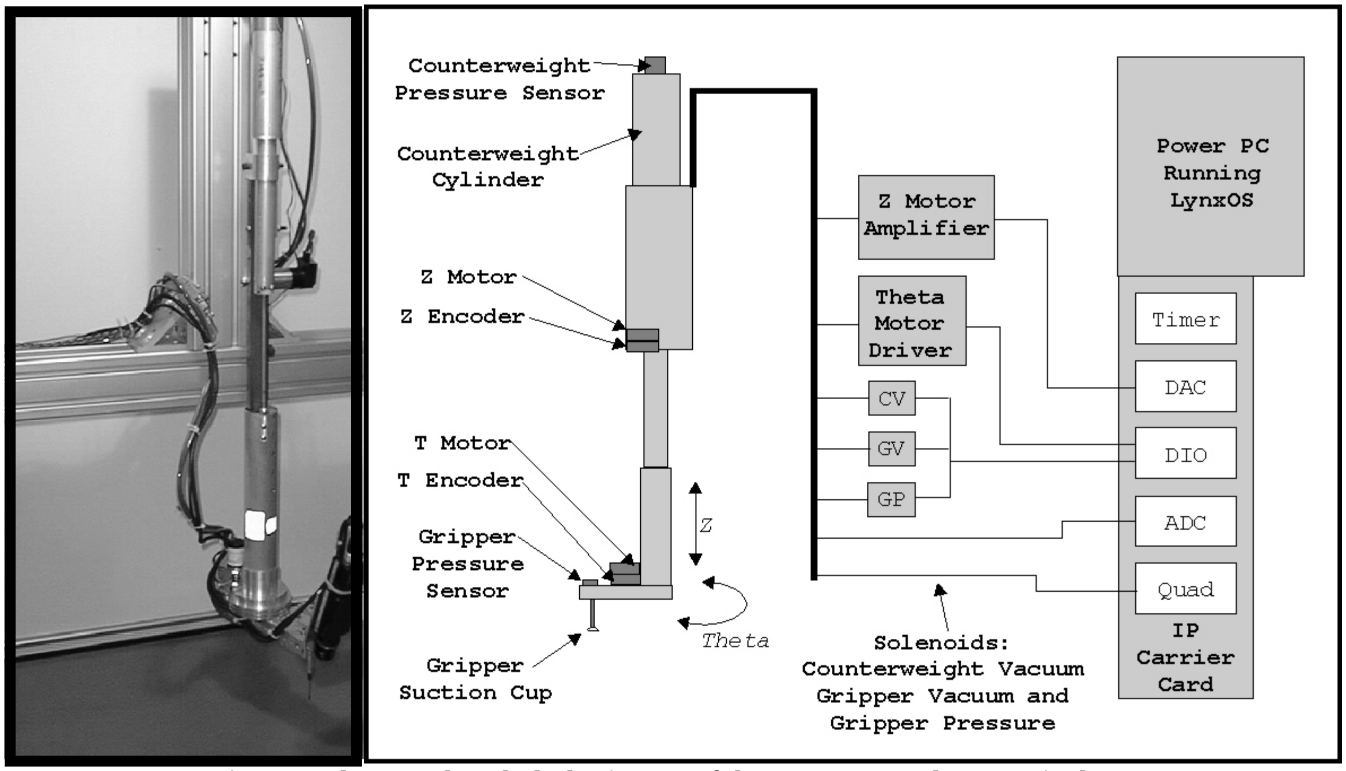

Figure 3: Photograph and Block Diagram of the Prototype Z-Theta Manipulator

5. CASE STUDY: Z-THETA MANIPULATOR

5.1 Minifactory Overview

Minifactory[11] is an on-going project within the Microdynamic Systems Laboratory in the Robotics Institute at Carnegie Mellon University. The goal of the project is to develop a modular system for automated precision assembly. The system concept centers around the operation of small table-top *courier* robots which carry subassemblies from one overhead processing station to another so as to realize the assembly of a product. It is envisioned that most processing stations will incorporate a *z-theta manipulator* for manipulation of parts. We have applied the UMOs described in this report to the control of a prototype z-theta manipulator.

5.2 Overhead Manipulator Hardware

Figure 3 shows a photograph and a block diagram of the major mechatronic components of our prototype z-theta manipulator. The control computer is a PowerPC running the Lynx real-time operating system. There is a carrier card plugged into the PC that allows the installation of up to 6 Industry Pack (IP) interface circuits. We utilize timer, digital to analog converter, digital input/output, analog to digital converter, and quadrature counter IP modules.

The z-axis has an associated servo motor, motor driver, and encoder with zero pulse. The theta-axis has an associated stepper motor[1], stepper driver with step and direction inputs, and encoder. There is a pneumatic counterbalance cylinder with an associated pressure sensor and vacuum solenoid. Gripping of parts is accomplished by an end-mounted suction cup which has an associated vacuum solenoid and pressure sensor for gripping parts, and a pressure solenoid for releasing parts. Note that the video camera in the photograph was connected directly to a video monitor for part viewing and is not considered in the following discussion.

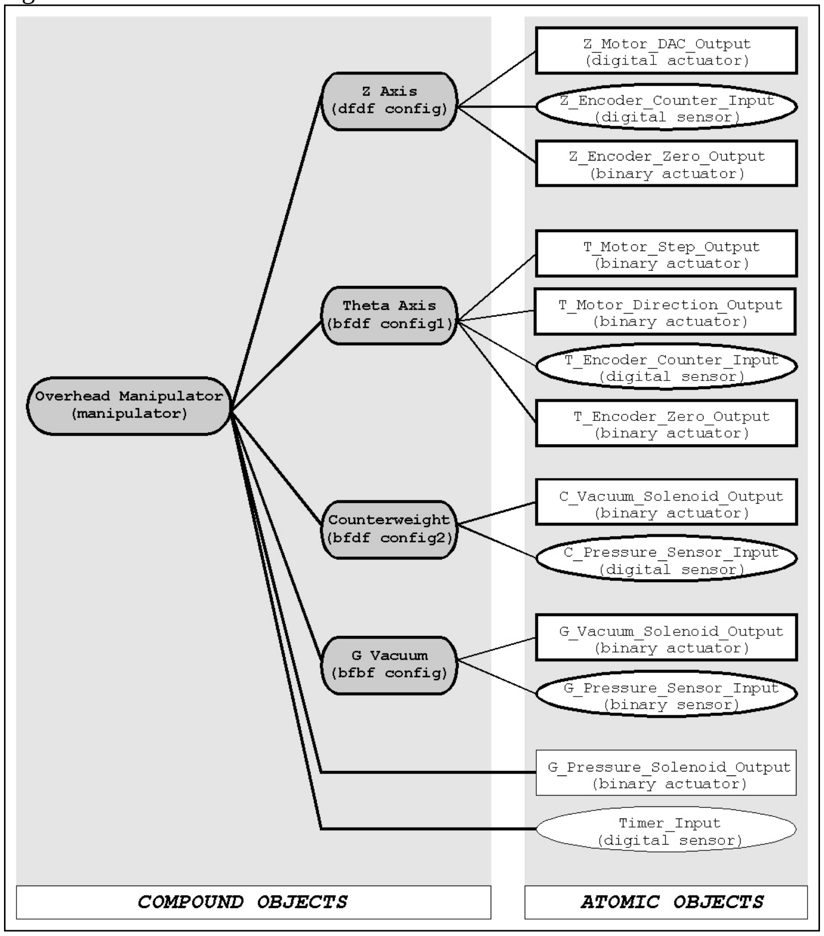
Overhead Manipulator (manipulator)
Z Axis (dfdf config)
Theta Axis (bfdf config1)
Counterweight (bfdf config2)
G Vacuum (bfbf config)
Z_Motor_DAC_Output (digital actuator)
Z_Encoder_Counter_Input (digital sensor)
Z_Encoder_Zero_Output (binary actuator)
T_Motor_Step_Output (binary actuator)
T_Motor_Direction_Output (binary actuator)
T_Encoder_Counter_Input (digital sensor)
T_Encoder_Zero_Output (binary actuator)
C_Vacuum_Solenoid_Output (binary actuator)
C_Pressure_Sensor_Input (digital sensor)
G_Vacuum_Solenoid_Output (binary actuator)
G_Pressure_Sensor_Input (binary sensor)
G_Pressure_Solenoid_Output (binary actuator)
Timer_Input (digital sensor)
COMPOUND OBJECTS
ATOMIC OBJECTS

Figure 4: Heirarchy of Mechatronic Objects for the Z-Theta Manipulator

[1] Because the prototype manipulator was intended as a testbed for control and application software development, this step motor and step/direction driver configuration was sufficient. However, a dc motor with PWM amplifier is planned for our second-generation manipulator design to enable more precise positioning.

5.3 Overhead Manipulator Objects

A straightforward mapping of actuators, sensors and their associated electronics and interface circuits to objects using the UMOs was performed and the results are diagrammed in Figure 4. Atomic actuators are diagrammed by rectangles, atomic sensors as ovals, and compound objects as rounded rectangles. It is evident that most of the objects are declared as atomic objects. Four compound objects are associated with the z, theta, counterweight, and gripper control groups. The remaining compound object represents the entire manipulator and aggregates both atomic and compound objects.

Notice that there is not a one-for-one mapping of hardware devices and objects. The *DIO-QuadratureCounter-Encoder* on the z-axis has two objects associated with it. One is a digital sensor representing the encoder count and the other is a binary actuator representing the control bit that allows the quadrature counter to zero when a zero pulse is detected by the encoder. Similarly, the *DIO-StepperDriver-StepperMotor* on the theta-axis requires one binary actuator to represent the step input and a second binary actuator to represent the direction bit. It would not be proper to declare the *DIO-stepperDriver-StepperMotor* as a 2-bit digital actuator because the two control bits do not represent the *numeric* value of a control signal. Notice that the counterweight pressure sensor is modeled as a digital sensor; whereas, the gripper pressure sensor is modeled as a binary sensor. This is correct because the counterweight pressure sensor has an analog output voltage that is read using the ADC. The gripper pressure sensor is a two-state threshold device that is read through one bit of the DIO.

The `Timer_Input` object is a good example of how to use a sensor object to synchronize the software execution to hardware that produces interrupts. The `IO_initialize` device driver segment for `Timer_Input` sets up a counter to produce interrupts every sampling period; in this case every millisecond. The `read` function of the `Timer_Input` can be called from the control software using the C++ line of code: `missed=Overhead_Manipulator.timer->read().` This line of code causes the `IO_read` device driver segment for `Timer_Input` to run. The `IO_read` segment blocks, allowing other processes to run, until an interrupt from the timer occurs, then it simply returns a zero to the calling software. If at least one interrupt has already occurred since the last time `read` was called, the function returns immediately with an integer representing the number of interrupts which were missed. Thus, even though there is no physical sensor involved in sampling period timing, a digital sensor object can be used to interface with the timer.

Listing 5: Declarations of Mechatronic Objects for the Overhead Manipulator

```
digital_actuator z_motor_dac_output(Z_MOTOR_DAC_OUTPUT,VOLTS,true,\
                                    0.0,10.0,10.0,(10.0/2047.0),0.0);
binary_actuator t_motor_step_output(T_MOTOR_STEP_OUTPUT,true,DOWN,UP,DOWN);
binary_actuator t_motor_direction_output(T_MOTOR_DIRECTION_OUTPUT,true,CW,CW,CCW);
binary_actuator z_encoder_zero_output(Z_ENCODER_ZERO_OUTPUT,false,DISABLE,ENABLE,DISABLE);
binary_actuator t_encoder_zero_output(T_ENCODER_ZERO_OUTPUT,true,DISABLE,ENABLE,DISABLE);
binary_actuator c_vacuum_solenoid_output(C_VACUUM_SOLENOID_OUTPUT,true,OFF,ON,OFF);
binary_actuator g_vacuum_solenoid_output(G_VACUUM_SOLENOID_OUTPUT,true,OFF,ON,OFF);
binary_actuator g_pressure_solenoid_output(G_PRESSURE_SOLENOID_OUTPUT,true,OFF,ON,OFF);
digital_sensor z_encoder_counter_input(Z_ENCODER_COUNTER_INPUT,MICRONS,true,\
                                       55000.0,129000.0,4.64357,-147871.0);
digital_sensor t_encoder_counter_input(T_ENCODER_COUNTER_INPUT,DEGREES,true,\
                                       -270.0,270.0,0.000529483,227.4438);
digital_sensor c_pressure_sensor_input(C_PRESSURE_SENSOR_INPUT,VOLTS,true,\
                                       0.0,1000.0,1.0,0.0);
digital_sensor timer_input(TIMER_INPUT,SAMPLING_PERIODS,false,0.0,99999.9,1.0,0.0);
binary_sensor g_pressure_sensor_input(G_PRESSURE_SENSOR_INPUT,true,FAIL,PASS);
dfdf_config z_axis(&z_motor_dac_output,&z_encoder_counter_input,&z_encoder_zero_output);
bfdf_config1
t_axis(&t_motor_step_output,&t_encoder_counter_input,&t_motor_direction_output,\
                                 &t_encoder_zero_output);
bfdf_config2 counterweight(&c_vacuum_solenoid_output,&c_pressure_sensor_input);
bfbf_config g_vacuum(&g_vacuum_solenoid_output,&g_pressure_sensor_input);
manipulator overhead_manipulator(&z_axis,&t_axis,&counterweight,&g_vacuum,\
                                 &g_pressure_solenoid_output,&timer_input);
```

5.4 Overhead Manipulator Software

Listing 5 shows the actual declarations of these objects. The compound objects in the system must be customized. One representative example is shown in Listing 6. The remaining compound objects are not included for lack of space, but can be reproduced easily using the one listed as an example. The compound objects only require three member functions:

`reset`, `update` and `sample`. Each of these functions simply calls the functions of the same name for each of its components. For example, the `update` member function for `Overhead_Manipulator` simply calls the update functions for its components which are or which contain actuators: `Z-Axis`, `Theta_Axis`, `Counterweight`, `G_vacuum`, and `G_Pressure_Solenoid_Output`. In this way, one call to the function `overhead_manipulator.update` will recursively update all the actuators in the system. Similarly, all of the sensors can be sampled using `overhead_manipulator.sample`.

Listing 6: One of the Compound Objects for the Z-Theta manipulator

```
class bfbf_config //binary_feedforward_binary_feedback_configuration
{public:
  binary_actuator* feedforward_device;
  binary_sensor* feedback_device;
  bfbf_config(binary_actuator* feedforward_device_parameter,\
              binary_sensor* feedback_device_parameter);
  update();
  sample();
  reset();  };
```

The complete set of manipulator objects forms the foundation for the control and application code for the system. One way to write modular control functions would be to develop the control functions as member functions of the appropriate mechatronic objects. For example, a function that steps the theta motor one step would be a member function of the object T_Motor_Step_Output. A function that returns the motor to a specific position would require access to the theta motor step and direction and the encoder, so it would be written as a member function of the theta_axis object. In this manner, the hierarchical organization of the mechatronic objects can be used to organize the control functions as well.

6. DISCUSSION

We have developed a set of four software objects, collectively referred to as Universal Mechatronic Objects, which provides the foundation for the development of control and application software for any mechatronic system. The flexibility of the UMOs to model a spectrum of actuators and sensors derives from the observation that, at the lowest level, actuators and sensors are accessed by the computer as memory-mapped IO devices. There is a close intuitive correspondence between actual hardware components and the mechatronic objects which facilitates rapid development and easy maintenance of the control software. The UMOs encapsulate the interface between the device-drivers and the control software providing an abstraction to the control developer which is hardware-independent, and thereby facilitating the portability and reuseability of the control code. We envision that this property may allow future systems to generate control code *automatically*. Consistent application of UMOs across several different mechatronic systems will allow software authors and maintainers to develop a familiarity with the objects which can further improve their efficiency, in contrast to the current tendency to develop custom code for each new system.

We have argued that the application of UMOs lead to many desired benefits during the development of a mechatronic system. Quantifiable results are difficult to obtain due to the nature of the benefits. In order to further this work, we are disseminating the UMOs to the interested users. We were unable to include the entire source code for the definitions of the UMOs in this paper because of a lack of space. Moreover, reuse of the code from hardcopy is time consuming and error prone. To facilitate the widespread use of UMOs we will make them and other associated code accessible over the internet at www.cs.cmu.edu/~muir/mechatronics.

7. ACKNOWLEDGEMENTS

This work has been supported by the National Science Foundation under grants DMI-9523156 and DMI-9527190. We would like to thank Ralph Hollis, Minifactory project lead and head of the Microdynamic Systems Laboratory, for his helpful suggestions and support over the course of this work. We would also like to acknowledge Ben Brown for the electromechanical design and construction of the z-theta manipulator and Al Rizzi for the PowerPC setup and the coding of the low-level device drivers for the z-theta manipulator.

8.REFERENCES

1. P.F. Muir, "The Growing Role of Mechatronics in System Realization," *SPIE's International Technical Working Group Newsletter: Robotics and Machine Perception*, Volume 7, Issue 2, August 1998, pp. 4-5.
2. D.G. Firesmith, *Object-Oriented Requirements Analysis and Logical Design*, John Wiley & Sons, New York, 1993.
3. T. Love, *Object Lessons: Lessons Learned in Object-Oriented Development Projects*, Sigs Books, New York, 1993.
4. B.R. Rao, *C++ and the OOP Paradigm*, McGraw-Hill, Inc., New York, NY, 1992.
5. B. Stroustrup, The C++ Programming Language, Third Edition, Addison-Wesley, Reading, MA, 1997.
6. G. Ericsson, "Functional Specification of Industrial Control Systems: An Object-Oriented Approach," *Proceedings of the Third IEEE Conference on Control Applications*, Glasgow, Scotland, UK, August 1994, pp. 1347-1352.
7. C.E. Pereira, "Applying Object-Oriented Concepts to the Development of Real-Time Industrial Automation Systems," *Proceedings of the Third Workshop on Object-Oriented Real-Time Dependable Systems*, 1997, pp. 264-270.
8. D. J. Miller and R.C. Lennox, "An Object-Oriented Environment for Robot System Architectures," *Proceedings of the 1990 IEEE International Conference on Robotics and Automation*, pp. 352-361.
9. J. Gausemeier, K.H. Gerdes and S. Leschka, "Cell Control by Intelligent Objects: A New Dimension of Production Control Systems," *Proceedings of the 1994 IEEE/RSJ/GI International Conference on Intelligent Robots and Systems*, Munich, germany, September 1994, pp. 47-55.
10. G. Glass and B. Schuchert, *The STL Primer*, Prentice Hall PTR, NJ, 1995.
11. R.L. Hollis and A. Quaid, "An Architecture for Agile Assembly," *American Society of Precision Engineering 10th Annual Meeting*, Austin, TX, October 1995.