An Object Oriented Generic Controller using CLIPS.

By Cody R. Nivens *

---

* Cody R. Nivens is a member of the Information Systems Staff of California Polytechnic State University, San Luis Obispo, California.

ABSTRACT

In today's applications, the need for the division of code and data has focused on the growth of object oriented programming. This philosophy gives software engineers greater control over the environment of an application. Yet the use of object oriented design does not exclude the need for greater understanding by the application of what the controller is doing. Such understanding is only possible by using expert systems. Providing a controller that is capable of controlling an object by using rule-based expertise would expedite the use of both object oriented design and expert knowledge of the dynamic of an environment in modern controllers.

This project presents a model of a controller that uses the CLIPS expert system and objects in C++ to create a generic controller. The polymorphic abilities of C++ allow for the design of a generic component stored in individual data files. Accompanying the component is a set of rules written in CLIPS which provide the following: the control of individual components, the input of sensory data from components and the ability to find the status of a given component. Along with the data describing the application, a set of inference rules written in CLIPS allows the application to make use of sensory facts and status and control abilities.

As a demonstration of this ability, the control of the environment of a house is provided. This demonstration includes the data files describing the rooms and their contents as far as devices, windows and doors. The rules used for the home consist of the flow of people in the house and the control of devices by the home owner.

INTRODUCTION

In the evolution of control mechanisms, it has become apparent that a higher level of knowledge of the system controlled must be embedded in the controller. This project uses the control of a house as an example of a knowledge-based controller. This is done by using the abilities of the CLIPS programming language to utilize user defined routines to input sensor information and to control external devices.

A real-time expert system can be defined as a system that decides in time to undertake a corrective action. Uses of such systems range from the home system described by this project to the control of nuclear power plants and space stations. Such systems have a set of common characteristics: compartmentalization; processes which run over minutes and hours; events which occur on a regular basis; exceptions to standard procedures which augment presently scheduled events; and a set of general rules on how operations in the controlled environment can be influenced by outside factors.

These principles illustrate the use of expertise: Specifically the body of knowledge acquired about the behavior of a complex system. The use of a rule-based knowledge system as a controller must have the following: the ability to control external devices; the ability to receive sensory information in a timely manner; the ability to make decisions within certain time limits; finally, the ability to expand as more knowledge of the behavior of the system becomes available. These principles are only a few that must be examined and met for such a controller to be effective.

The home enviroment is becoming a laboratory for the design of user-friendly control systems. Such systems are programmed in one of several procedural oriented languages and as such they are difficult to expand to meet the needs of the user. A solution to this problem is the use of real-time expert systems. These systems provide the logic in a style that is easy to update and understand. A carefully crafted expert system could be updated and changed by the home owner with little need for their understanding of the rule system.

This paper discusses the combining of CLIPS with objects defined in C++ to create an intelligent controller. The C++ objects define what is controlled. It is mated together with the CLIPS expert system, with CLIPS supplying the expertise for the control of the object. This is done by a loop mechanism which alternates between CLIPS, the C++ objects, and an interrupt information structure. CLIPS controls the object via external functions which access the objects controlled. The user interface employs the objects as a selection mechanism and the assert routine of CLIPS.

OBJECT ORIENTED PROGRAMMING

Object oriented programming is ideally suited for use in intelligent controllers for several reasons. There are several reasons for this. First, an object oriented programming language allows for the creation of an abstract data type. Second, components of a program can inherit functions and data from other objects allowing for the reuse of previous code. Finally, an object oriented programming language provides for the use of polymorphic characteristics. The abstract data type is the key feature of an object oriented syste.

An abstract data type is called a class. A class is composed of the data structure associated with the implementation of the data type and a set of member functions which manipulate that data structure called member functions. There are several advantages to creating a new data type: the hiding of the implementation of a design from its user, encapsulation of both the data and the code that manipulates it, and the restriction on access to the data reducing inter-module dependences. Member functions allow limited access to the data of a class. These are messages that the class accepts for manipulating itself. The function passes the parameters necessary to complete the desired operation. Member functions can be overloaded by using the same name with different parameters. This feature allows descriptive function names to have different routes to the same service.

Inheritance and polymorphism are interrelated in their uses. Inheritance allows both code reusability and the derivation of new data type types that share both the code and data of its base. Polymorphism uses this ability to create derived classes which use functions of the base class and redefine functions in the derived class. Functions which can be redefined are called virtual functions. The virtual function differs from the normal function in that the binding to the function occurs at run time as opposed to the static binding at compile time. There are two major uses for this feature. First, the redefined function of the derived class is used when the base class calls the function. Second, a pointer to a base class can be used on a derived class with the functions redefined by the derived class being used. This allows the calling program to use a derived class without knowing what it is. For example, an array of components which having the same base class can all be sent the same function call even though each component in the array is a different derived class.

For example, consider a set. The implementation of a set in the C++ language consists of two classes as defined in figure 1. The first class is a set element. The second class is the set itself. Two types of set elements are defined in figure 2 to show how the inheritance and polymorphic abilities of C++ work. The main program and output is defined in figure 3. Note that 'a' is said to be an instanciation of the set class. This is similar to saying that x is an instanciation of an integer, but is not an integer class.

DEVICE CONTROLLERS

Computer based controllers fall into three broad categories. First, the group of controller are those controllers that are based on a clock signal. These controllers deal with the use of a set sequence of events that are triggered when a predetermined time arrives. An example of this is a steel mill which heats a piece of metal for a predetermined length of time. A second type of controller is based on sensory input. These controllers must provide a response based on input from the environment of a device. Examples of this type of controller are the closing of valves based on the level of liquid in a tank. The last type of controller is interactive. These controllers generally deal with human input and have their responses geared towards the average person using the device. An automated teller machine is an example of this type of controller.

INTELLIGENT CONTROLLERS

An intelligent controller will be defined as a controller that has the ability to arrive at decisions based on external facts and internal rules of the behavior of the system being controlled. To illustrate such a controller, a model of how the controller relates to the controlled component is needed. The simplest way to achieve this is to consider the controller as an indivisible computer. The inference engine is the cpu, the rules are the programs, and the fact lists are the data. I/O for such a computer consists of external assertions of facts and the execution of commands from the consequent portions of rules.

The use of a central processor for the CLIPS engine is a very useful metaphor. The Rete algorithm uses tokens of the changes in working memory to communicate which rules may fire. Such a system is similar to the concept of an associate memory system. All changes within the memory system happen at one time. The tokens affect only those rules that use the changed component of working memory. Such a scheme allows for a large number of rules and facts to be compiled into a network whose access time is dependent on the changes in working memory.

The model of the cpu would have to be extended to include the use of interrupts. In CLIPS, interrupts could be handled by rules that are fired by the assertion of a specific fact. The chain of events that follows from the interrupt can be determined by the precedence of the rules. The use of the salience feature allows for the running of priority tasks based on interrupt information. Each set of interrupt rules would have a salience level associated with it. It should be noted that the CLIPS system handles input from the interrupts, not the interrupts themselves.

Programming the Device Controller

Programming the CLIPS machine for the use of several independent processes involves little change in method from conventional programming. The major difference between normal programming and this model is the use of a set of rule chains to determine the "program." The need for scheduling, enqueing or dequeing for resources, or rendezvousing between tasks is eliminated. All these things are handled by the working of the Rete mechanism. Two tasks which have independent chains of inference can perform a rendezvous via the assertion of a common fact.

For example, the standard consumer/producer problem can be defined in CLIPS by two rules as shown in figure 4. The producer/consumer cycle starts with an assertion of the specific producer facts and the start fact for the producer rule. The cycle between the producer/consumer is controlled by two facts which are asserted when the particular phase of the cycle is done. Such a system does not have the ability to enque messages, but such abilities can be accessed via an external procedure.

Interrupts

Interrupts and device input are handled in a similar manner. The use of the add_exec_function allows a user defined routine to be used between the firing of rules. This function then has the option of asserting information based on the state of an interrupt or device. The control of such assertions can be handled by two routines defined by the define_function routine. One function enables interrupts from devices and external interrupts. A second function disables the asserting of new facts. A supporting function returns the state of interrupts. Interrupt precedence can be controlled via the salience clause of a CLIPS rule. This allows specific interrupts to have control of the system while they are working. An example is shown in figure 5.

Input/Output

Traditional device input is handled by the add_function routine of CLIPS. This function allows for the creation of a routine which can be used in the RHS of a rule. The function defined would then assert a fact based on the responding device. Output is handled in a similar manner: the defined function would take a multi-variable pattern and consult the appropriate component being controlled.

THE GENERIC CONTROLLER

The generic controller is an object which uses an expert system to provide control to some other object. The controller class has the following components: a CLIPS expert system, a component to control, a simulation to run the component through, an alarm manager for time signals and alarm activations, a command object to pass commands between CLIPS and the controlled object, windows to display output for the user, and a set of I/O ports for information on the component controlled and through which to control the component.

The basic use of the controller consists of loading the information on the windows, the ports, the component information, the simulation information and the files that the CLIPS system will use for a trace of all its output, as well as the rules and data of the controller and the application. Next, either the controller is run in real-time mode where the alarm manager and ports deal with the real-time and hardware of the system, or the controller is run in simulation mode where the time and port values are artificial.

In either case, the controller goes into a loop where the following events occur endlessly. First, the CLIPS expert system is executed for a set number of inferences (rule firings.) Second, if a command was executed by the executive function then the status is updated. Third, the keyboard is checked for user input. If input is found, it is passed to the controlled component to interpret. If the interpretation returns a command string, the string is asserted into CLIPS after the current time is attached to it. Next, the sensor inputs are checked for new data. If input is present, it is asserted into CLIPS after the time is stamped onto it. Finally, the alarms are checked and the time is updated if necessary.

THE CLIPS CLASS

The CLIPS class is not an implementation of the CLIPS expert system, but is an interface to the C routines that define the CLIPS system. The encapsulation of CLIPS in a C++ class has enabled the restriction of the many available routines that provide access to the CLIPS environment. The member function of the CLIPS class provides for the following areas of access. First, the embedding functions of clear, reset, execute and load are given standard names and definitions of their use.

The CLIPS class also provides for the use of I/O routers. These functions allow for access to external I/O devices. The use of this function requires that the functions passed to the I/O router not be a member function of a class. The reason for this is that while the address of the member function is known, the instanciation of the class it is being used by is not known. As such, the I/O router functions are defined as friend functions to the controller class.

The next area that the CLIPS class provides a common interface for is the use of executive functions. The executive function is one that is called by the interpreter of CLIPS rules between rule firing. In this project, the executive function is responsible for asserting sensory information if it is present.

The next member function that the CLIPS class contains is concerned with defining a function that CLIPS can call from the right-hand side of a CLIPS rule. This function can do work outside of the CLIPS environment, possibly returning a value as a predicate function. There are three functions defined in this project: do_command, seek, and set_alarm.

The interaction between CLIPS and external routines are defined in two member functions: The first asserts a string into the CLIPS environment, and the second loads a command object with the parameters passed to a function when it is called by a CLIPS rule.

The last set of functions in the CLIPS class are involved with debugging and status display. These routines deal with the activation of watches on facts, rules and activations. They also provide functions for the display of the CLIPS fact environment and the current agenda of rules to fire.

THE COMPONENT CLASS

The component class is the class which describes the object being controlled. This class provides a generic holder for information on how a system relates to itself. This scheme is a hierarchical system. Objects at one level only access those at a lower level and the parent of the present object. Access across branches of the component tree are not possible in this system. A component provides an object display, I/O, and relational information.

The display information of a component is divided into four parts. The first part is a window display of the contents of a component in a window. The second part is a display of the status information about the component. The next area is a display of the related objects of the component. This part consists of an overlay which fits the related objects into a cohesive whole. The last area consists of the display windows and the index to the window in which the overlay and related components of the component are displayed.

The I/O information consists of several values. The input port id determines which related object is the next component in the component-path name of the input item. If there are no related objects then the value from the port is the state of the device or sensor. The output value, the command or value related to the place of the component in the system begin controlled, is

sent to the output port. If there are no related objects, the output value is the state of the component. The I/O ports are an array of ports that are used for input/output operations. These allow for an index to determine which input port and which output port should be used. The I/O ports are used by the interrupt mechanism to establish an interrupt path to a component. This is done by enqueing the id of the component in the set of related objects of the parent component.

The related object information consists of the related objects, their number and which are currently selected. This information is used to create command strings that are asserted into the CLIPS system. The related object information identifies which is the master (root) component and which component is active (being selected from.)

The use of individual I/O ports, command levels and display windows allows the programmer to create generic components that are independent of the device being controlled. For example, the application of this project is a house controller. In the test case, there are 3 rooms, 10 lamps, 13 outlets, 12 sensors, and 12 command components. All can be represented by generic components. All I/O in the system is done by the generic component; no further programming is needed. A draw-back is that the number of components goes up with an increase in command complexity with any device. The simple solution to this is to create new device components derived from the base component.

SUPPORTING CLASSES

The alarm manager class has four major functions. First, it is responsible for the time and date clock. Second, it holds the times of alarms that are active in the CLIPS environment. Third, when an alarm occurs, the alarm manager asserts a time fact into CLIPS for the time of the alarm. Finally, the alarm manager class is responsible for the time stamp when an event occurs.

The command class acts as a data carrier for communications between CLIPS and the component. There are two parts to a command: the count of lines in the command, and the lines themselves. The command class is defined as an array of strings. The dimensions of the array are dynamically enabled when the class is instanciated. It must be noted that the CLIPS version used in this project has multiple field variables containing extra lines of information, specifically, the fact name-field (the first field in the fact.) Hence, the offset must be one greater than the position of the field in the multivariable of the CLIPS rule. This can be used to allow one routine to interpret many commands, as the command is always the first field.

The port class defines an input/output medium. The port can either be used for real I/O or for simulated I/O. Real I/O is

device and implementation dependent. The simulation of the port input is done via an index that the port acquires along with a simulation when it is instanciated. This id is passed to the simulation which returns -1 if either the index is lower than the ports simulated or there is no input for the port ready. The ports used for the house application are shown in figure 6. Interrupts use the ports to signal that a value is present. This is done by the interrupt routine which calls the component. It changes the state of the component and creates an interrupt trail via a member function of the parent of the component.

The simulation class contains an array of values that are assigned to ports dependent on the time that the simulator has for the next input. The first member function deals with the loading of the simulation values from an input stream. There are two functions which deal with stepping the simulation and testing if the simulation is done. Two further functions deal with returning the simulation time and the simulation value given a port index. The private variables of the simulation define the number of simulations, the offset for the port index, the current simulation time, and the index of the next simulation event.

SYSTEM RULES AND FACTS

The system rules are divided into four areas: changes in sensory information, time and date maintenance, alarm durations, and activation of alarms.

The first set of rules in the system CLIPS file deals with sensory information. This section is divided into two parts. The first deals with the rules involved with the processing of sensory input. There is only one rule: sensor-reset. This rule resets the sensor input states when the sensor cycles from ON to OFF or OFF to ON.

The second set of rules dealing with sensory information seeks status of components in the system. There are three rules in this set: seek-status, status-seek, and reset-seek. Seek-status is used to reset the knowledge system given existing state facts. This allows for the periodic checking of the consistency of the knowledge base against the controlled component. Status-seek processes the results of a seek operation by creating a state fact. Seek-status and status-seek work with a control fact: seek-state. Seek-state carries a list of selector elements, which quizzes related objects and their descendants for their status. Reset-seek retracts the seek-state fact if no other rules are activated by the fact. The structure of the system facts are listed in figure 7.

The second part of the system rules is composed of guidelines related to the maintenance of time and date. When the date changes at midnight, the alarm manager asserts the new date.

This assertion causes the rule change-date to fire. This rule asserts seek-state on all components and process-alarms to set up the alarm manager for the next 24 hours. The reset-time rule removes the time fact if no other rules are activated by it. The time fact is asserted by the alarm manager when an alarm occurs.

The third set of rules are those involved in processing alarm times. There are three rules. Process-alarms is activated by the process-alarms fact asserted by the change-date rule. Set-alarm-time sets the time of a newly activated alarm. Reset-process-alarms removes the process-alarms fact if no other rules are activated by it. A more complex system of rules would process alarms on an hourly basis.

The next section of the system rules is concerned with rules which govern the use of durations. Durations are alarms which run from one time to another. This section is divided into three parts. The first part is the rule set-duration. This rule is activated by process-alarms asserted by the rule change-date. The second part consists of the rules start-duration and reset-start-alarm. Start duration fires when the alarm created by the duration is activated. It asserts start-alarm fact containing the id of the alarm activated. This is asserted for application rules use when alarms are activated. Reset-start-alarm removes the fact if no other rule is activated by it. The last part consists of the rules end-duration and reset-end-duration. End-duration removes the alarm associated with a duration. It fires when the duration reaches its end. It also asserts the end-alarm fact with the id of the duration associated with the retracted alarm. Reset-end-alarm removes the fact end-alarm if no other rules are activated by it.

The final set of rules consists of the rules for the firing of alarms. There are nine rules which correspond to the types of alarms. All alarms have the following in common: an id, a type, a possible repetition count, a date and time to fire, and information specific to the application which is used to command the controlled component. The alarm types are listed in figure 8. Alarm fact structures and constants are listed in figure 9.

THE APPLICATION

The application of this project consists of a house controller. The basic design focuses around the use of the X-10 house controller. X-10 is an industry standard for the control of components in a home. The application consists of a three room building. Each room has at least one door, one or more windows, lamps and outlets. For each room, there is an overlay file, a list of devices in the room as well as CLIPS facts on the room. The house as a whole also possesses an overlay.

The controller is used in a command mode by selecting which room to work in. Next the type of device to control is selected.

The device is then selected. Finally, the command to perform on the device is selected. When this is done, a command is sent to the CLIPS controller. The controller in turn sends a command to the component to perform the operation.

The application and the controller have performed well in simulation runs. It will soon be implemented in a model system consisting of the basis house that is now defined along with X-10 controlled devices. The outcome of this implementation will be presented at the CLIPS Users Conference.

HOUSE RULES AND FACTS

The house rules file is divided into three parts. The first part deals with door direction and specification information. The second part deals with room and house occupancy. The last part contains exception rules for possible error conditions.

The door direction rules are outside-door-dir and door-dir. Outside-door-dir is concerned with determining if a person is entering or leaving the house. Door-dir determines which room a person is entering and leaving.

The next set of rules deal with house and room occupancy. The first rule is changing-rooms which adjusts the appropriate room occupancy counts. The next rule is person-entering-house. It adjusts the house occupancy count and the room being entered or left.

The last set of rules contain two exception rules. The first is person-too-many-room. This rule resets the room count and issues an exception message to standard out. The second rule is person-too-many-house. This rule resets the house and appropriate room count and issues an exception message to standard out. Figure 10 shows the house controller fact structures.

SUMMARY AND CONCLUSIONS

The use of CLIPS as a real-time controller in a house has be examined. The CLIPS expert system is suited to this work because of its abilities to define external functions and executive functions which allow the insertion of interrupts into the working storage of the system. This allows the CLIPS system to be viewed as a computer with programs, interrupts, and input/output capability.

The use of rule-based systems as opposed to procedurally-based systems gives a programmer greater control over the logic embedded in a system. As the logic of a system goes beyond a certain limit of comprehension, rules for clarity become necessary. Traditional control systems in conventional languages are based on simple formula describing the system. In an application such as a home, a descriptive formula is all but impossible. Yet, it is possible to describe the behavior of the system in pseudo-English. This pseudo-English allows the programmer to develop rules that describe the behavior of the system. These rules are then given directly to the controller without need for additional programming or development.

The use of an object oriented programming language allows the creation of descriptive fact structure related to the component being controlled. C++ is a language which provides such capability in a familiar setting. A programmer familiar with C will have little difficulty improving or adding code. This reduces the cost of development of new projects, and their maintenance once they are in operation.

Intelligent controllers are a natural extension of Artificial Intelligence into the fields of conventional programming and control. Embedded systems may one day have the ability to control and learn from previous conditions and actions. Research into such systems will prove to be profitable and stimulating. CLIPS is an excellent tool with which to conduct such research as it is written in C, which combined with C++, allows for programmer involvement in the development of the rules and structure of the application.

Figure 1 - Set Classes

```
class set_element {
friend set;
private:
        set_element * next; // pointer to the next element in the set.


public:
        set_element();
        // Effects: Creates a set element.


        virtual print();
        // Effects: Prints the set element's contents.
};


class set {
private:
        int size;                  // number of elements in the set.
        set_element * elements; // The elements in the set.


public:
        set();
        // Effects: Creates a set.


        add(set_element* a);
        // Requires: A set element to add to the set.


        print();
        // Effects: Prints the contents of the set.
};
```

Figure 2 - Derived Set Classes

```
class card : public set_element {
private:
        int value;
        int suit;


public:
        card(int v,int s);
        // Requires: A value and a suit.
        // Effects: Creates a card with value of suit.


        print();
        // Effects: Prints the value and suit of the card.
};


class toy : public set_element {
private:
        char* name; // Name of the toy.
        char* color; // Color of the toy.


public:
        toy(char* n,char* c);
        // Requires: Name and color of the toy.
        // Effects: Creates a toy.


        print();
        // Effects: Prints the toy.
};
```

Figure 3 - The use of the Set class and its output

```
#define DIAMONDS 1
#define HEARTS 2

main()
{
        set a;

        card d10(10,DIAMONDS);
        card h1(1,HEARTS);


        toy doll("doll","blue");
        toy ball("ball","green");


        a.add(d10);
        a.add(doll);
        a.add(h1);
        a.add(ball);


        a.print();
}
```

OUTPUT

```
green ball
ace of hearts
blue doll
10 of diamonds
```

Figure 4 - Comsumer/Producer Rules

```
(defrule consumer
    ?f<-(consume $?a)
=>
      .
      .  misl processing
      .
    (retract ?f)
    (assert (produce a)) )

(defrule producer
   ?f<-(produce a)
     .
     . Specific producer info
     .
=>
   (retract ?f)
     .
     .  misl processing
     .
   (assert (consume $?a)) )
```

Figure 5 - Interrupt Rule

```
(defrule fire-rule
        (declare (salience 10000))
        (fire ?room)
        (sprinklers ?room $?sprks)
=>
        (sound-alarm)
        (bind ?i 1)
        (while (< ?i (length $?sprks))
               (do_command ?room (member ?i $?sparks) ON)
               (bind ?i (+ ?i 1))
        )
)

Interrupt asserts (fire room1).
```

Figure 6 - Input/Output Port Definitions

INPUTS

Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7
NULL	X10	House	Room	Device Type	Device	Command	Dim Value

OUTPUTS

Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7
NULL	X10	NULL	NULL	NULL	NULL	NULL	NULL

Figure 7 - Application Independent Fact Structures

The following information consists of the structure of the facts that are used by the controller. These facts are generic to all applications that run on the controller. In the house rule, data and alarm files, their use is further illustrated.

Application Facts: These rules deal with the contents of application specific information. The format of the rule does not change only the contents of the $?info field.

```
(action ?action-type $?info ?state ?time)
(sensor $?info ?state ?time)
(state $?info ?state)
(status $?info ?state ?time)
```

Where:

?action-type	- Action description: Usually user defined based on sensor information sensor reset is signified by break in ?action-type field.
$?info	- Application specific information
?state	- State location is in (i.e., on, off, 0, 1, etc.)
?time	- Time status was returned from controlled object.

Figure 8 - Alarm Types

ALARM EVENT TYPES

TYPE	DESCRIPTION
one-time	Fires on specified date and time and is removed from the system.
daily	Fires every day.
week-day	Fires Monday through Friday.
week-end	Fires on Saturday and Sunday.
weekly	Fires each week on the same day.
biweekly	Fires on the first week day and then 3 days later.
monthly	Fires each month on the same day.
every-day	Fires every specified number of days.
every	Fires every specified number of seconds.

Figure 9 - Fact Structures and Time Constants

```
     Alarm Facts:
        (alarm ?id ?event-type ?event-repetition ?year ?month
               ?day ?time $?info)
        (alarm-mark ?id ?event-type ?event-repetition ?year
?month
               ?day ?time $?info)
        (date ?year ?month ?day ?day-of-week ?julian-date)
        (duration ?id ?from ?to)
        (new-date ?year ?month ?day ?time ?day-of-week
               ?julian-date)
        (time ?secs)

Where:
        ?id                 - Alarm id - either number or character
                              string.
        ?event-type         - Determines how and when alarm is
fired.
                              See above table for event types.
        ?event-repetition   - Determines frequency of event.
                              Used by
                                 weekly     - Day of week to
                                              activate alarm.
                                 biweekly   - First day of week
                                              to activate
                                              alarm on.
                                 every-day  - Number of days till
                                              next alarm.
                                 every      - Number of seconds
                                              till next alarm.
        ?year               - Last two digits of year.
        ?month              - Month id based from zero.
        ?day                - Day of month.
        ?time               - Time of day in seconds.
        $?info              - Application specific information.
        ?secs               - Number of seconds since midnight.
        ?from               - Time in seconds to start alarm.
        ?to                 - Number of seconds to allow alarm to
run.
        ?day-of-week        - The day of the week with Sunday as 0.
        ?julian-date        - Days from beginning of year to
present.

Constant Facts: These facts are constant through out the life of
an application and from application to application.

        (biweekly-map 1 2 3 4 5 6 0 1 2 3)
        (month ?month-id ?month-name ?days-in-month)
        (week-days 1 2 3 4 5)
        (week-end-days 0 6)
        (year-length 365)
```

Where:

?month-id	- Id of month (January - 0)
?month-name	- Jan, Feb, etc.
?days-in-month	- length of month in days

Figure 10 - House Controller Fact Structures

The following consists of the structure of the facts that are unique to the house controller application.

```
(door ?house ?door ?room1 ?room2)
(door-sensor ?house ?room ?sensor ?door-type ?door)
(outside-sensor ?house ?room ?sensor)
(people-in-house ?house ?number)
(people-in-room ?house ?room ?number)
(window-sensor ?house ?room ?sensor ?window)
```

Where:

?house	- House id door is in
?door	- Door id
?room1	- Room 1 id
?room2	- Room 2 id
?sensor	- Id of sensor
?door-type	- Door type: door, outside-door
?number	- Number of people
?window	- Window id