IsoMax™ Documentation

Introduction

IsoMax™ is a programming language designed for special problems requiring Isostructure. Most embedded and real time applications require Isostructure. These problems cannot be easily implemented in other non-Isostructured languages. The Isostructured problems have previously been solved only with extensive reliance upon multitasking. Compilers, parsers, communications controllers and user interfaces are also best done with isostructure techniques.

The creation of Isostructure concepts is a significant advance in programming technology, particularly to the field of Computer Science. Several previously difficult areas of endeavor are now trivially accomplished with Isostructure techniques. In particular, problems requiring concurrency are easily mastered.

The word, Isostructure, has two roots: Iso, meaning equal or "on the same level," and structure, from the current usage of structure in the Computer Science field, derived from the important concept of "structured programming."

Background

Previously, the two concepts of program organization were separated into two arenas: programming language and operating system. The structure was chiefly localized in the "structured programming language." The control of entry into individual programming segments, corresponding to the Isolayer, was the domain of the operating system. While suitable for some data processing environments, this separation was a false dichotomy leading to sticky problems, particularly in real time programming.

As a result there has been a unnecessary tension between Computer Science and Engineering. Engineers tried to create the Isolayer of Isostructured programming themselves. In their programs they used GOTO's in high level languages, or abandoned the restrictions of structured languages by using assembly language and direct control of interrupts. For the programmer from the Computer Science community, the code that resulted appeared to be horrifyingly unstructured, badly behaved and difficult to maintain. For the engineer, code written within the limitations of structured style was indeterminate, slow and wasteful of system resources, requiring large overheads for embedded operating systems.

Fortunately, the creation of Isostructure brings the two communities happily together and benefits both. Isostructure removes one portion from conventional, structured-programming style, and places an Isolayer on top of independent, structured threads. An

Isolayer is a layer of interconnection at equal level. The Isolayer connects the structured threads in a method similar to that previously done by the much-more-complex, multitasking Operating Systems. All repetitive entry into the independent threads is moved to the Isolayer. This alters the structured programming technique of looping, and removes the problem of Program Counter Capture experienced with that method.

Program Counter Capture occurs when the processor's Program Counter is used to retain the state information of a program implicitly, rather than explicitly. When state information is held in the Program Counter, the Program Counter must be rigorously maintained intact. The thread cannot release control of the Program Counter, or the state information is lost. The only way to regain control of the Program Counter for use on other tasks or threads is to preempt the task and store the Program Counter and register states in stacking operations, performing a full context switch. This is why a multitasking operating systems are usually required.

Unfortunately, conventional structured programming relies exclusively on Program Counter Capture to hold state information. Conventional structured programming offers no Isolayer, and few of the tools necessary to create it, so the only acceptable methods to create loops cause Program Counter Capture.

Isostructuring separates Isolayer state information and conventional structure into distinct entities. State information is stored explicitly and does not depend on the Program Counter. Structure within the thread is still the domain of the Program Counter. Without Program-Counter-capturing looping structures, threads remain deterministic and well-behaved segments. State information is the domain of the Isolayer. State information determines which threads are active and forces deterministic and regular activation of pending active threads.

Prospects

Isostructure techniques hold the keys to advancing through the previously closed doors of several highly desirable areas of technology.

Fault tolerant technologies are more easily achieved if state information is not kept solely in the CPU's Program Counter Register. When state information is held explicitly, rather than exclusively (thereby elusively) in the Program Counter, the crash of the processor is not necessarily tantamount to system failure. The Isostructure can be re-entered and program flow will be directed to the threads needing service.

Similarly, multiprocessing is possible. Active threads are known, and available processing power can be directed or "parceled out" to active threads as needed. It is even possible to have multiprocessing where the processors are non-similar. The different processors can coexist peacefully in a single system. Since state information is not in the domain of any single Program Counter, any processor with time available can service the threads, so long as they are compiled in non-processor-specific tokens.

Isostructure goes farther than any other language has ever done before. It is possible to compile some structures written with Isostructure techniques into hardware. Hence the line between software/firmware/hardware is significantly blurred. The same program can be written as software, moved to firmware, or extended into hardware.

The technique of separating state information from the Program Counter also offers potential advances in the area of Artificial Intelligence and Expert Systems. The firing of inference engines are essentially the recognitions or expansions of state information. As such, untangling the processor's Program Counter from the state information can simplify understanding of context, the fundamental problem standing in the way of achievement in AI.

Fundamentals

The enforcement of structured programming techniques some twenty years ago made certain areas of programming difficult. Primary among them was the area of real time programming. It is difficult to find a good definition of real time programming in publication. Recognition of the problem is the first step toward solution. Therefore the following definition is offered.

A real time program is one that waits. Any program that does not wait on something is not a real time program. Any program that waits on something, other than the program itself, is a real time program.

Almost all embedded system programs are real time in nature, which is to be expected. Surprisingly, large programs which are seldom suspected of having any real time component, often do contain small embedded real time programs in them. For instance, a word processor is seldom classified as a real time program. Yet, it waits on the user's input before taking any action. The user interface of a word processor is, therefore, a real time program. The same is true for most user interfaces. They wait, therefore they are real time. Understanding the effects of waits on programming is the first key to understanding Isostructure.

A good word processor not only waits for a user's first input, it also checks often to see if there is new input, even when it is already actively processing the previous input. A word processor that makes the user wait, while repaginating for instance, does not have the feel of a user friendly system. So all tasks the wordprocessor does, that may take more than an imperceptible amount of time, must be written in a way that allows multiple threads of concurrent operation to be processed. This means all wordprocessors must create Isostructure and support real time programming, whether done intentionally or not. The same is true for most user interfaces. Understanding the need not-to-be-trapped by waits is the second key to understanding Isostructure.

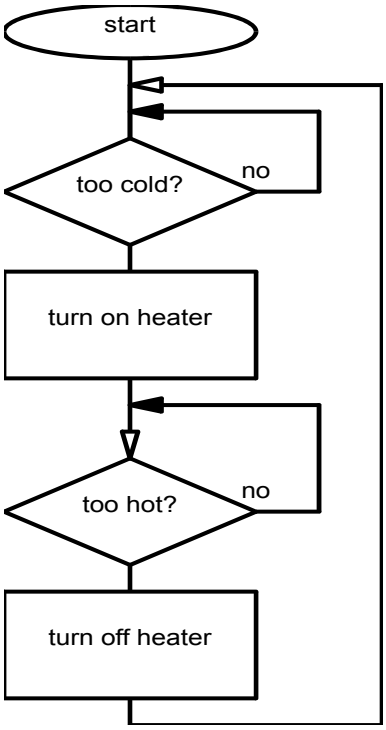
Once multiple "threads which do not wait" are allowed to be active, a system must be configured to cause entry and re-entry of these threads as necessary. Transitions from one waiting state to another require more sophistication than conventional structured programming methods or bifurcation offer. Both these instances call for more fundamental capability than simple IF-ELSE-THEN structures provide. CASE statements are often employed here, although they seldom produce anything more than program-managed bifurcations. This is not surprising, since structured program advocates have held a prejudice against any structure containing more than a single bifurcation. A multiplicity of possible branch transitions must be accommodated to create true Isostructures. Understanding the need for polyfurcation in programming is the third key to understanding Isostructure.

To fit these three keys into practice, it will be helpful to examine the current programming paradigm further.

Problem

The limitations of conventional structured programming can be seen in the following example, illustrating the three keys: wait states, the need to avoid Program Counter Capture, and polyfurcation.

Imagine one of the easiest embedded programming examples, a thermostat. One analog input is read. One output is manipulated. When a the analog signal is too low, the output is turned on. When it is too high, the output is turned off. A flowchart illustrating one approach for programming the task is shown here:



While the conditional test of temperature is not "too cold" the Program Counter is captured in a backwards loop, waiting on a change. Sometimes this is called "spinning on a bit" or in this case "spinning on a (analog) conditional." While the Program Counter is captured in this loop, no output is generated. When the conditional test "too cold" is first true, the Program Counter Capture loop is terminated. Next, an action is taken. The heater is turned on. The Program Counter is then allowed to advance. The next Program Counter Capture loop is entered. While the conditional test of temperature is not "too hot" the Program Counter is captured in a backwards loop, waiting on a change. When the heater has done its job, and the temperature is finally high enough, the Program Counter Capture loop is terminated. An action is taken; the heater is turned off. The Program Counter is then allowed to advance. The program flow takes the Program Counter back to the original Program Counter Capture loop. This continues in an endless loop.

As shown, the thermostat is a very easy program to write, and indeed, if the one thermostat is the only problem to be addressed by the processor, can be written just that simply. Conventional structured programming tools masterfully fit the need, up to this point. In pseudocode the program might look like this:

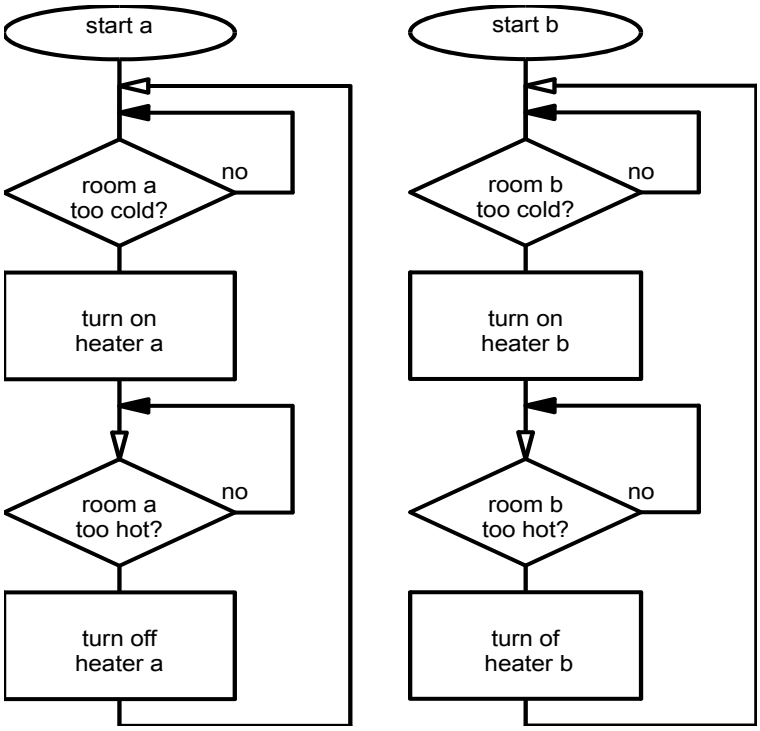
```
start:
begin
     begin
          not_too_cold?
     until
     turn_on_heater
     begin
```

```
            not_too_hot?
        until
        turn_off_heater
    again
```

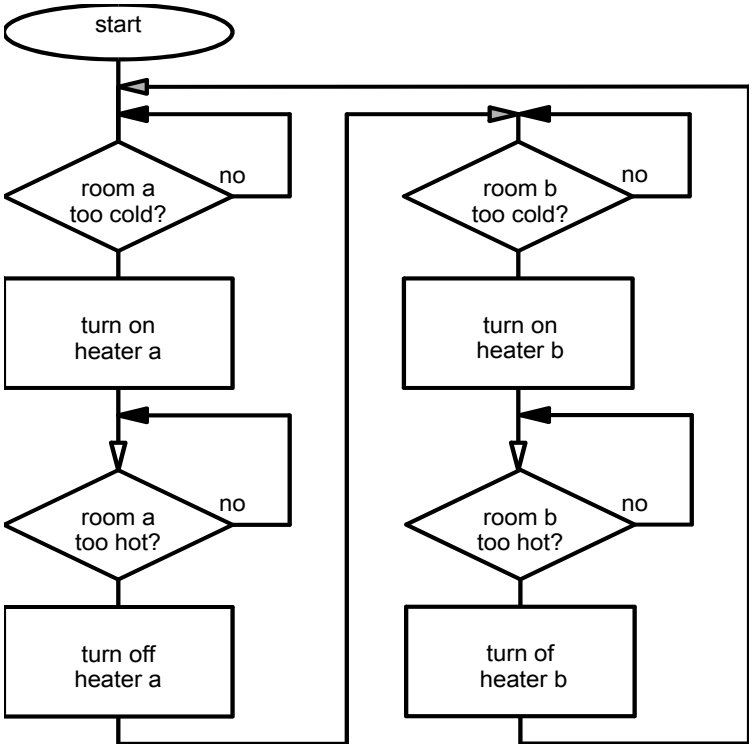
No GOTO's are needed. No polyfurcations are required. Only well structured programming techniques are used. Both engineers and computer scientists see the program as well written and well behaved. In practice, the generated code would perform well, although the problem is a bit oversimplified. A breif explanation follows.

From a Control Science perspective, this situation is very minimally specified. The dead band is not specified. The amount of "hunt", undershoot and overshoot are not determined. Still, even if there is no dead band, that is, if the "too cold" temperature is the same as the "too hot" temperature, the thermostat will still operate. The heater will short cycle according to the resolution of the temperature sensor. So while it is "too cold" the heater is on. While it is "too hot" which is functionally equal to "not too cold" the heater is off. Spreading a positive difference between "too cold" and "too hot" conditions makes a dead band, which reduces heater cycling. However, only when the dead band becomes negative does the algorithm have bad characteristics, short cycling (turning on and off) the heater as fast as the program can run.

So, for a single task, there is no problem with conventional methods. The problem starts when the task is only slightly more complicated. To bring this situation to light, imagine one processor controlling the temperature in two rooms. Essentially, the same algorithm can be used twice if a multitasker is employed.



The length of the program is very small, perhaps a few hundred bytes. A multitasker, by comparison, is one or maybe even many orders of magnitude larger. To the engineer, adding thousands of bytes to the short program to achieve preemptive multitasking seems an overhead that should be designed out. Here is where the trouble starts. As a first approach the engineer may try to string the two programs together to create one.

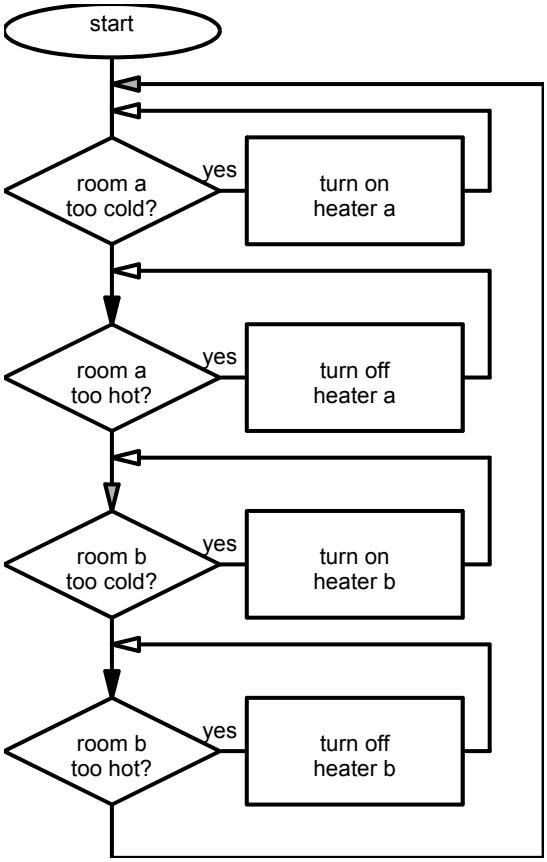


While properly structured from a programming view point, this program is not functional. The two rooms do not function independently. While Room A is not too cold, the first Program Counter Capture loop prevents any other conditions from being tested. So, Room B will go unattended if Room A is still not too cold. Room B can be too cold for a long time. When Room A is finally too cold, heater A is turned on. Room B is still left unattended until Room A is finally too hot. Then Room B is serviced. While Room B is heating from its long state of being too cold, Room A is unattended. Even if Room A cools off to the point it is too cold, it will not be serviced again until Room B is too hot. This algorithm is not workable. The pseudocode shows the shortcoming as well.

```
start:
begin
     begin
          room_a not_too_cold?
     until
     turn_on_heater_a
     begin
          room_a not_too_hot?
     until
     turn_off_heater_a
```

```
        begin
             room_b not_too_cold?
        until
        turn_on_heater_b
        begin
             room_b not_too_hot?
        until
        turn_off_heater_b
    again
```

For the second attempt the engineer may abandon the first algorithm in preference for one that has a different structure. To move toward this end, the original program may be rewritten with opposite senses for each branch.



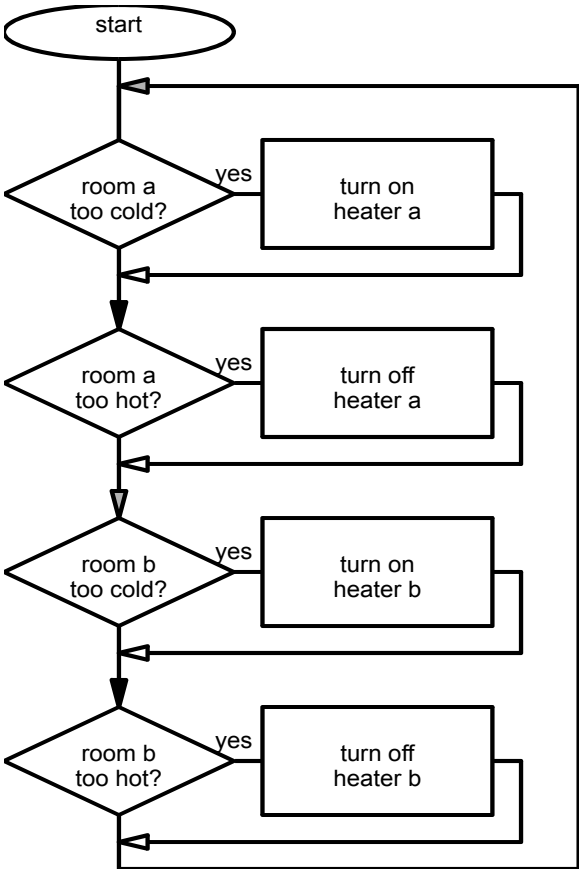
This algorithm is no more functional than the previous implementation, and in some ways is less so. It has all the bad habits of the former. It has an additional bad habit of turning the heater on when it is already on. If the heater is simply controlled by a binary output, such as a port line connected to a Solid State Relay driving an electrical heater, there is no harm in repeatedly turning on the output. If the heater is a gas heater with an ignition sequence, and there is more complexity in turn_on_heater_b, this algorithm is unacceptable. The pseudocode follows.

```
    start:
```

```
begin
     begin
          room_a too_cold?
     while
          turn_on_heater_a
     repeat
     begin
          room_a too_hot?
     while
          turn_off_heater_a
     repeat
     begin
          room_b too_cold?
     while
          turn_on_heater_b
     repeat
     begin
          room_b too_hot?
     while
          turn_off_heater_b
     repeat
again
```

The one advantage of looking at the problem this way, however, is that the conditionals "string" together and suggest the sense of a bifurcation tree (or in advanced programming models, a case statement). At this point the clever engineer recognizes the backwards branches as the source of the program's problem. Recognition of the problem as Program Counter Capture is the first step toward solution. The clever engineer may attempt to create a scan loop. A scan loop is a programming technique using software polling. Conditions are checked in a round-robin fashion and only those conditions needing service are executed.

To create the scan loop, the first approximation requires elimination of the backwards branches which create Program Counter Capture. The following flowchart has been changed from the previous implementation only by redirecting the backwards branches "forward" to allow the program counter to freely run around the scan loop without being captured in any particular place waiting, or spinning on a condition.

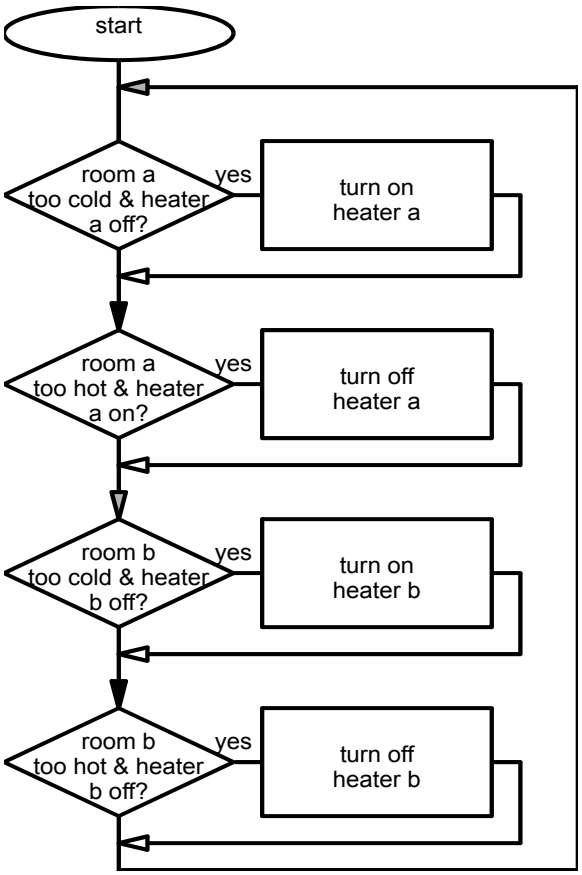


While there are some bad habits inherent in this implementation, it performs better than the previous designs in one important aspect. If either room becomes too cold, service for that room begins immediately.

The bad habit of turning the heater on when it is already on, is still present in this design. Also, the system is more sensitive to dead band limitations. If there is no dead band, that is, if the "too cold" temperature is the same as the "too hot" temperature, the thermostat will not operate satisfactorily. The heater will short cycle as fast as the program can run. The problem is the same as if there were a negative dead band. The heater is slammed on and off as fast as the possible. If careful restraints are not put on the operator, an equal or negative spread between too_cold? and too_hot? conditionals can accidentally put the system into a mode that violently chatters the heater's power. The pseudocode for this implementation follows.

```
start:
begin
     if   room_a too_cold?    then
turn_on_heater_a
     if   room_a too_hot?     then turn_off_heater_a
     if   room_b too_cold?    then turn_on_heater_b
     if   room_b too_hot?     then turn_off_heater_b
again
```

The bad habits turning on a heater that is already on, can be cured by adding a test to the conditional to see if it is off. If it is already turned on, it is not turned on again.



The pseudocode for this implementation follows.

```
start:
begin
     if   room_a too_cold?    and  heater_a_off?
     then      turn_on_heater_a
     if   room_a too_hot?     and  heater_a_on?
     then turn_off_heater_a
     if   room_b too_cold?    and  heater_b_off?
     then turn_on_heater_b
     if   room_b too_hot?     and  heater_b_on?
     then turn_off_heater_b
again
```

This is the first time state information has entered the programming example since the Program Counter Capture loops were eliminated. Previously, the loops with backwards branches captured the Program Counter. While the processor was limited to only executing instructions in the loop, the state information about the process under control was keep implicitly in the processor's Program Counter. Since the backward branches

were eliminated, there was no obvious state information kept by the program. If the output state of a heater can be "read back" this state information can be used in program control.

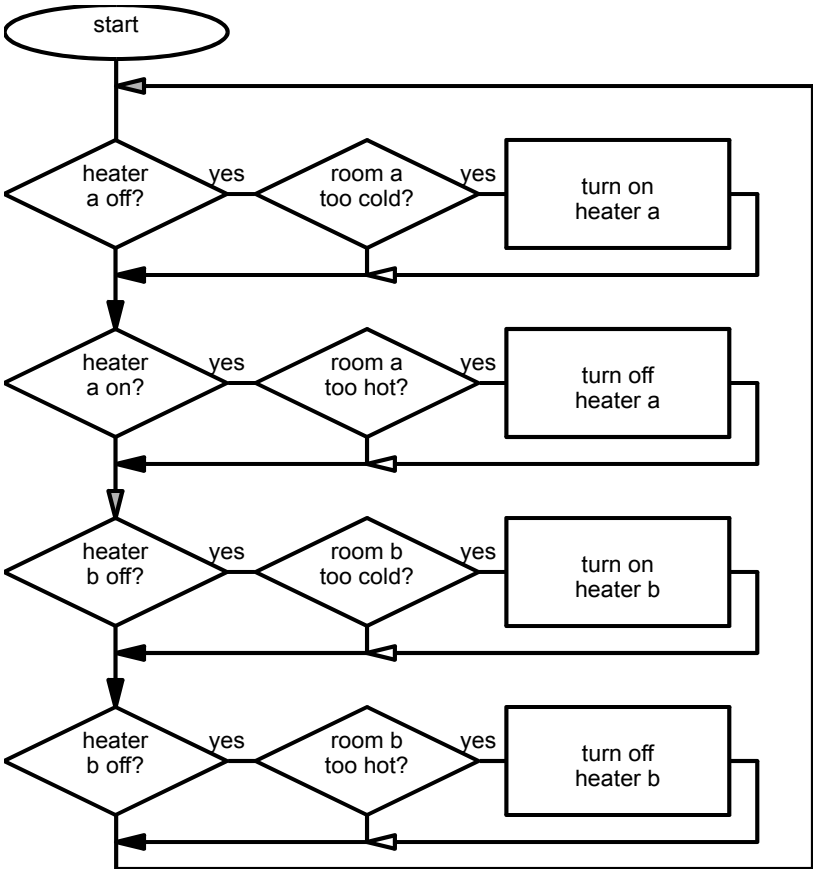
The bad habit of turning on the heater when it is already on can be overcome if the heater's state information is retained. Testing the state information to see if the heater is already off before turning it on, and testing to see if it is on before turning it off, makes for a better behaved program. The operation of a program designed in this manner is closer to the first example of a single thermostat.

While the read back of the output is one way to retain or, more correctly as described in this case, recapture state information, it is less direct than simply saving the state information in memory explicitly. The actual details of the storage, whether variable, vector, table index or whatever, are less important than the conceptual achievement of recognizing the need for such explicit storage.

State information is the essence of the class of programs examined here. It represents the history of the program's execution. It holds the context against which decisions are made. All the information needed to take future actions based on previously occurring conditions are part of the state information. State information is absolutely necessary and vital to a control or real time program. Recognizing it and applying it correctly produces Isostructure.

The remaining bad habit of short-cycling the heater can be reduced by using state information to create Isostructure. At the same time, the overall performance of the program can be improved. Once Isostructure techniques are applied, all the positive aspects of the original, simple thermostat programs will be recovered.

Applying Isostructure will be best understood if taken in two steps. The following example closely resembles the previous example, except the state information becomes the primary test. Since the prevailing style of programming has been to hide the state information in the Program Counter, it not surprising that the state information might not be considered the primary information to process on every scan. It is, however, more important than other conditionals. The reason will become obvious momentarily. First examine the example with the state information as the primary conditional:



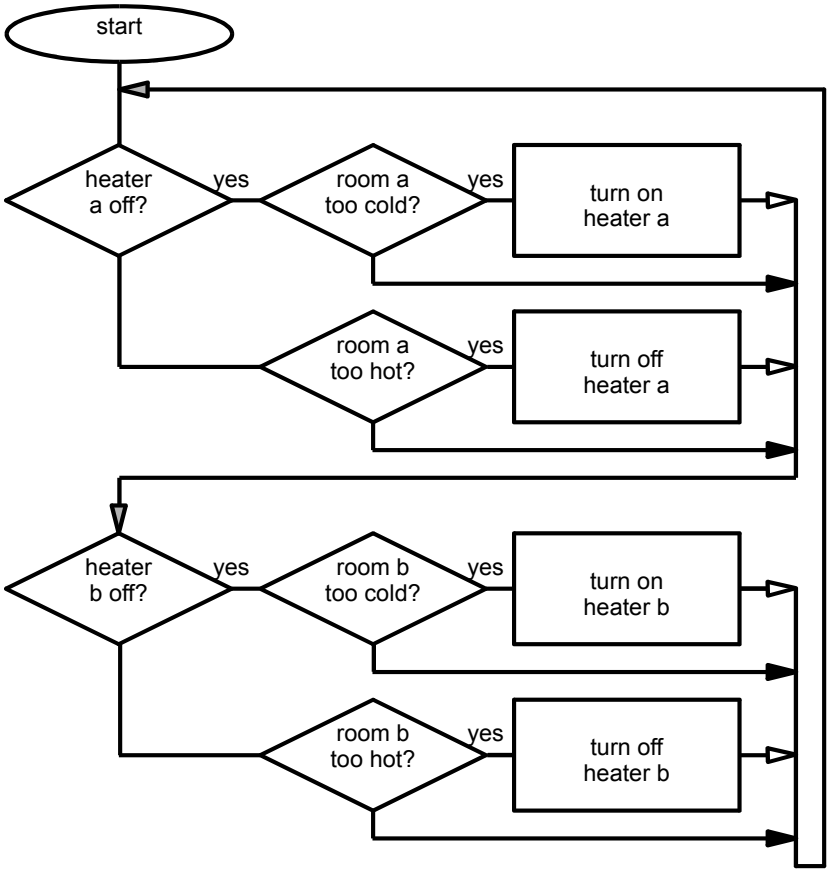
The pseudocode for this implementation follows.

```
start:
begin
     if   heater_a_off?
     then if   room_a too_cold?
          then      turn_on_heater_a
     if   heater_a_on?
     then if   room_a too_hot?
          then turn_off_heater_a
     if   heater_b_off?
     then if   room_b too_cold?
          then turn_on_heater_b
     if   heater_b_on?
     then if   room_b too_hot?
          then turn_off_heater_b
again
```

The structure can obviously be simplified. Separate "if's" are used for complementary conditions. Simplifying the structure would make the program faster and more efficient. However, this first step was taken to make easy visualization of the one-to-one correspondence of conditionals with the previous example. It turns out, the key to

correcting the dead band behavior also lies in the path of simplifying the structure. Isostructure techniques must be applied to overcome the dead band behavior.

The following example uses Isostructure. Bifurcation, the simplest instance of polyfurcation, occurs based on the state information for each thermostat. This example closely resembles the previous example. The duplicity of state conditionals have been replaced with a single instance. The structure has been simplified, thereby. The return paths of conditionals are gathered differently as well, since there is no need to do all conditionals under all conditions. The same state information was tested twice in the previous example. In this example a single test of state information replaces the two separate tests. The program flow is bifurcated to the following conditions, according to which one is of interest in the active state. Following the bifurcation of program flow, only one of the two possible temperature conditionals is evaluated. (For later reference, notice: based on that test, another bifurcation occurs.)



The pseudocode for this implementation follows.

```
start:
begin
     if   heater_a_off?
     then if   room_a too_cold?
          then      turn_on_heater_a
```

```
        else if   room_a too_hot?
             then turn_off_heater_a
        if   heater_b_off?
        then if   room_b too_cold?
             then turn_on_heater_b
        else if   room_b too_hot?
             then turn_off_heater_b
    again
```

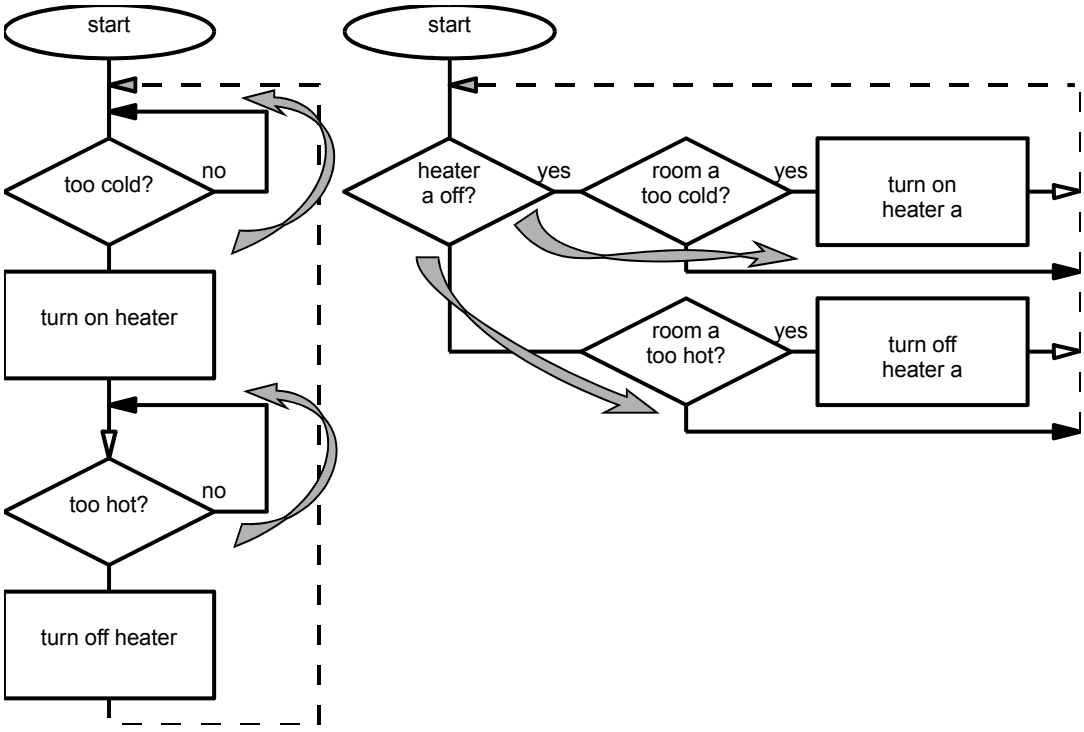
All of the undesirable habits of the intermediate versions have been overcome by this implementation. The scan loop immediately serves either room needing heating turned on, or heating turned off. The heaters are turned on only if they are already off and need to be turned on. The heaters are turned off only if they are already on and need to be turned off. The deadband is the same as the original single thermostat program. Finally, the two-thermostat program has been restructured into a workable structured model without using multitasking.

Below, a single thermostat's portion of the previous example is extracted and placed side by side with the original thermostat flowchart. Notice the Program Counter Capture loops are gone. State information directs the program flow to the appropriate test condition. There are two branches of program flow on each pass through the program. The two flows are represented in the illustration by gray, curved pointers.

If there is no heating being done, the conditional test for "heating needed" is evaluated. When it is true, heating begins. No more testing of "heating needed" will be attempted. As much as can be done to achieve heating is already underway and there is no need to check the corresponding condition.

If there is heating being done, the conditional test for "no more heating" is evaluated. When it is true, heating is stopped. Then, no more testing of "no more heating" will be attempted. As much as can be done to stop heating has already happened and there is no need to check the corresponding condition.

In the original thermostat flowchart, there are also two "branches" of program flow. These two flows are represented in the illustration by gray, curved pointers, as well. In the original, these follow the Program Counter Capture loops, as enforced by conventional structured programming techniques. The program is prevented from evaluating both sets of conditionals, but not by explicit state information. Instead, the program execution is not allowed to reach the inappropriate conditional by capture of the Program Counter in a backwards loop constantly reevaluating the active conditional.



In the Isostructured version, the Program Counter is not captured. Program flow falls directly through the structure on each pass. As a result, the program is deterministic, and has known characteristics and delays for each path. It can be grouped with other Isostructured problems in a scan loop. Each problem is entered on the same level, at the Isolayer. The result is manageable multitasking without a multitasker.

Comparing the two flow diagrams, it should be clear that the original approach is the most common way of approaching such a problem. The Isostructured approach is certainly readable in this specific example, but the need for the Isostructure is not intuitively obvious. From the perspective of years of work done with structured programming tools and limitations, something about the look of the Isostructured approach "feels" odd.

If a simple problem looks "odd" a complex one must seem absolutely "foriegn." It is fortunate there were only two states in the thermostat problem. More complex multitasking problem cannot easily be diagrammed with conventional structured programming tools. The Isostructure techniques required in this implementation were limited. Only bifurcation was used. Problems with more than two wait states require polyfurcation. While polyfurcation can usually be represented with nested bifurcations, readability is quickly lost. The resulting diagram is difficult to follow and the intent of the program is exceedingly obscure.

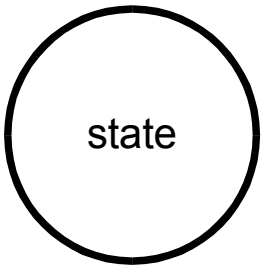
When larger problems are written with conventional structured programming tools using Isostructure techniques, the bifurcations compound, splitting into finer and finer branches. The program begins to resemble a period doubling chart from Chaos theory. The greater the level of bifurcation, the closer the flow chart looks to a chaotic system.

(As a side light, there are interesting speculations which can be imagined concerning chaos in real time systems. A real timer program can be "chaotically" programmed using accidental Isostrucutre techniques as has been previously common, or run chatoically with "randomly" interrupted Program Counter Capture loops which has been the other popular approach. One method puts the chaos in the compile time, the other, in the run time. A better method would be to avoid the chaos from the beginning. Polyfurcation replaces nested bifurcations.)

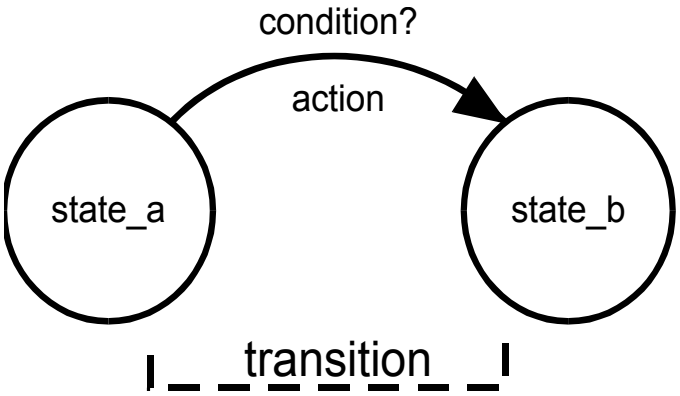
Clearly a better diagramming tool is necessary, and a better paradigm is in order. Just such a better tool exists in state diagrams as will now be explained.

Description

Based on the previous two section, it should now be obvious that Isostructure has two easily identifiable features: States, which represent programmed-waits explicitly, and polyfurcation, which allows multiple transition paths into and from the waits. Diagrammatically, states can be portrayed as objects and transitions as the relation between objects. (Isostructure, therefore, implements the equivalent of hardware State Machine Diagrams in software programming. An approximation of the parallelism of hardware is achieved in software using Isostructure.)



When in a state, as its Latin roots {*status and stare: to stand}* *imply static condition, no outputs or actions are taken. The only activity occurring in a wait state is the periodic checking of conditions which are antecedent and necessary to cause termination of the waiting state. (This parallels the synchronous clock in hardware state machines, which causes conditions to initiate transitions.) A state can be identified by its quiescence. In a state, no outputs are changed. If outputs need to be changed, the wait state is no longer valid. If no outputs need to be changed, the wait state is valid. States in the new paradigm stand in place of Program Counter Capture loops in the previous paradigm.*



Transitions in the new paradigm stand in place of processing boxes between Program Counter Capture loops in the previous. All processing actions take place during a transition from one state to another. Transitions have four components. The first component is the state with which they are associated. The second is the condition under which they cause transition. If the condition is met, the current wait is no longer valid. The transition becomes selected. The third part of a transition is the action. (This is the Mealy state machine model.) Once a condition is valid, actions must be performed. Often this requires processing and output manipulation. This action has no waiting component. If there are any waits built in it, it is improperly fashioned. The transition's action is done as fast as the available hardware can be made to run. The final component of a transition is the vector, or destination of the transition. As soon as possible, all processing will be accomplished. Re-entry to a condition of real time waiting follows. Therefore, a transition always ends in another wait state.

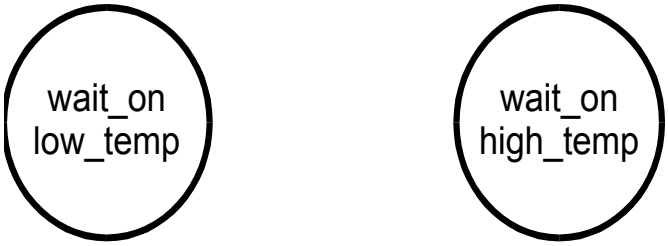
By assigning all computing to its component levels as waits or transition, the entire Isostructure of a problem can be laid out. By doing so, all Isostructure components will be separated from the processing portion of threads. Threads can then be programmed with conventional structured programming techniques. Program Counter Capturing loops and waits will be removed. Therefore properly defined Isostructured programs will have well-behaved threads.

Diagramming

Using state machine diagrams to define real time problems is quite easy. The first step is to identify and define all unique waits. The states can be drawn as circles. States should be uniquely named. The name of this state is written inside the circle.

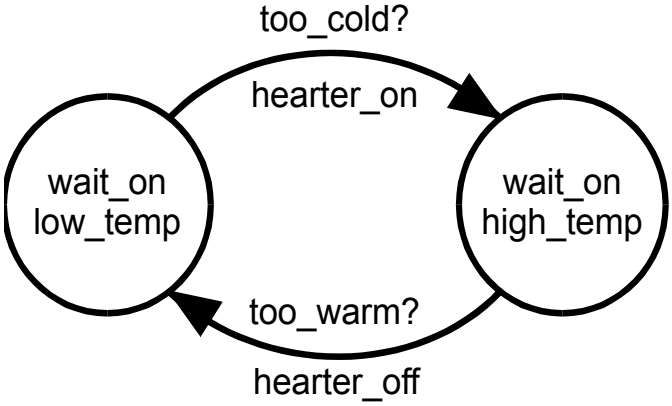
In the case of the single thermostat problem, two wait states are required. First the two wait states must be named. A newcomer might assert, "One state is 'heater_on'. The other state is 'heater_off'." This is not a good start. These definitions refer to the states in terms of the actions associated with the (previous) transition which put the machine into this wait state. For a second attempt, the beginner might suggest, "One is 'too_cold?' and the other is 'too_hot?'." This is only slightly better. These definitions refer to the states in

terms of the conditions associated with the (next) transition which take the machine into the another wait state. The experienced programmer recognizes a wait state is not necessarily defined by the actions which cause entry or the conditions which cause exit. It is the nature of the wait itself which should be named. The programmer experienced in Isostructure and state machines might suggest names such as these, "One state is 'wait_on_low_temperature' and the other is 'wait_on_high_temperature'."



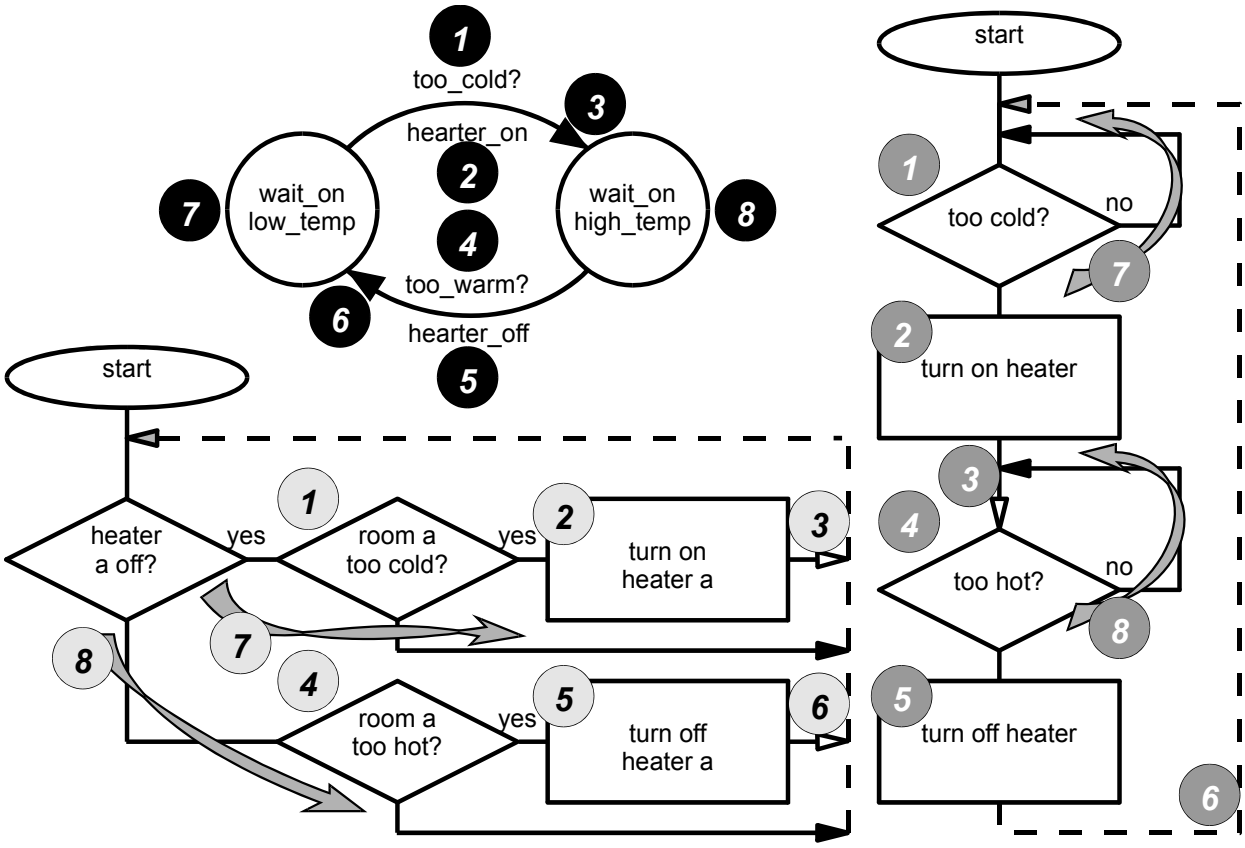
In the case of the thermostat, each state has only one entry and one exit. Consequently, these suggested state names resemble the conditions of exit. However, in cases where there can be multiple exit transitions, it is not likely this resemblance will be so obvious. For instance, a wait_for_key state might do different things if the key were a numeral, versus a "enter" key, versus a "delete" key, versus a "backspace" key, etc. State names should describe the nature of the wait as best possible. For the sake of the underlying computer language, the names must be unique.

Once the states are defined as objects, the relationship between the objects are specified in the form of transition. The thermostat has two transitions.



Each of the four components of the transition must be specified. The originating state and the destination state are illustrated graphically. The condition and the action are entered as text written along the vector of the transition. The condition is normally placed above the transition line, and ends with a "?" to designate its conditional nature. The action is normally placed below the transition line, and is a statement of action. In the above diagram, the thermostat program is as well defined as it was in the flowcharts of the preceding sections.

In fact, there is a high degree of correlation between the two diagrams. (This should not be surprising. They represent the same problem, differing only in the nature of the paradigm.) The two diagrams are shown side-by-side below with the commonalities identified.



This comparison dramatically shows, all the components of the previous flowcharts are covered by the state diagram. The second thing the comparison shows, which comes through with startling impact, is how much simpler the data looks in the state diagram format. It is difficult to believe those few elements used to define the thermostat problem in the state diagram format contain all eight components found in the other examples. Even the diagram's added numbering seems to clutter and overpower the beautifly-simple underlying state diagram. The paradigm of the state diagram used to describe applicable Isostructure problems is remarkably comfortable, concise and easy to understand.

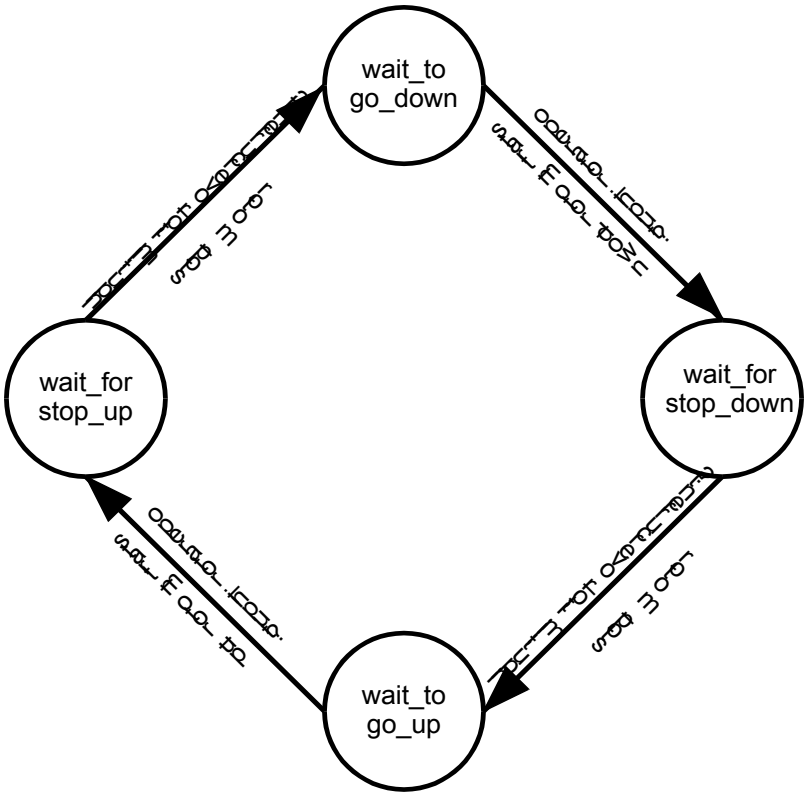
The thermostat program represented graphically in this paradigm consists of two circles and two arrows, with some text annotations. The non-Isostructured flowchart has two conditional diamonds, two action boxes, over six flow path arrows, and text annotations (and a start box). The Isostructured flowchart has all that plus an additional conditional diamond and two more flow path arrows. Clearly the state machine paradigm has a more succinct "feel" about it. Further, it better describes the problem. It names the wait states, which were only implied with the Program Counter Capturing loop of the non-Isostructured version of the flow chart, and further obscured by the Isostructured version.

The intellectual advantage of recognizing the wait state is a bit like the discovery of "0". In evaluating ancient cultures, their ability to recognize "0" as a numeral is considered a high water mark. Neither the Greeks nor the Romans understood the concept of "0". The Arabic and Mayan cultures were among the first to use it. By naming the condition where nothing exists, higher level math can be accomplished. In direct analogy, recognizing the wait state in a program as a place where "'0' computing" takes place, allows higher-level programming. In this sense "'0' computing" means no outputs or changes of state occur. Only the conditionals are tested during waits. While the conditionals are false, nothing else meaningful happens. "0" actions are taken. So specifying wait states in programming is like using "0" in math, very essential to advanced processes. To illustrate this point, two increasingly complex applications will be detailed, 1) a garage door opener and 2) an electronic keypad lock.

In the simple example of the thermostat, the process was a simple sequencer. Sequencers are a subset of state machines. Each state in a sequencer has only one exit and one entry. The flow from state to state is circular (i.e. in sequence). It is relatively easy to program sequencers with conventional structured programming methods, using Program Counter Capture loops between the steps (states) of the sequence. Even a simple sequencer with more than two steps can become devilish to flow chart using Isostructure techniques. These problems need polyfurcation and structured techniques only offer bifurcation.

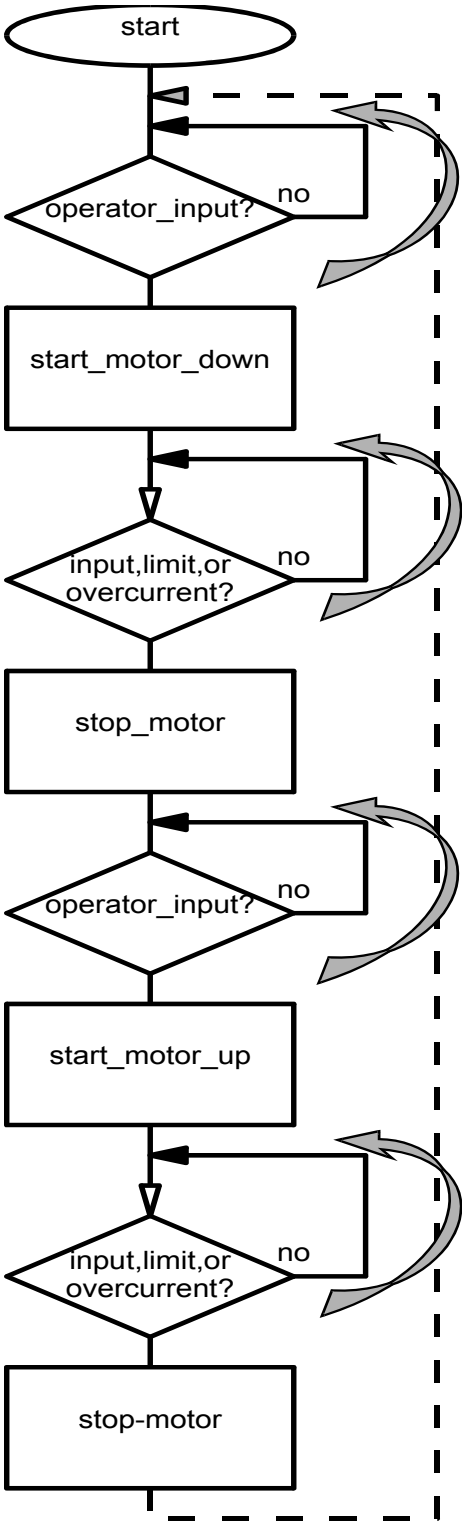
A garage door opener is a good example of a four step sequencer. To pick a starting point, consider the situation with the door up. What is it waiting for? an input to tell it to go down. When the input comes, it starts the motor moving down and then waits. In this state it waits for another input. This can be the "already down" limit sensor, or the current sensor on the motor saying the system is blocked or jammed, or the operator's pressing the button again indicating a change of mind. When any of these conditions occur, the motor is stopped and the system waits for another signal to start up again. When that signal is received, the motor is started up and the system then waits for the "already up" limit signal, or the current sensor on the motor saying the system is blocked or jammed, or the operator's pressing the button again. At this point, the sequencer has gone full cycle. The process is ready to start over again.

The state diagram for a garage door opener is shown here.



It is quite succinct. The information is terse. The problem is well represented. In short, the paradigm works well for representing sequencers.

The flow chart (nonIsolstructured) for a garage door opener is shown here.



Certainly this flow chart is readable. However, many more graphical elements are required to represent the problem this way. Hence, the ratio of graphic element to problem illustration is bigger. The problem is, thereby, obscured by the flowcharting paradigm. For a single tasking problem, flowcharting obscures the problem only slightly.

Adding isostructure to the flowchart further obscures the problem. While the Isostructured flow chart for a garage door opener is not too complicated to be drawn, it is larger and therefore not conveniently shown here. Four additional nested conditionals would need to be added, to choose which state the garage door opener was in. These conditionals would in turn lead to the conditionals already shown in the non-Isostructured version. Since the garage door opener example is a sequencer, rather than a more complicated state machine, the Isostructured flow chart would still be regular and perhaps readable, though less so than the non-Isostructured flow chart and certainly much less than the state diagram model, which is inherently Isostructured.

The second example, an electronic keypad lock, is not so easily represented with conventional structured techniques. The electronic keypad lock is a state machine, rather than a simpler, sub-class sequencer. The state machine diagram for a three digit entry lock is shown below.

–

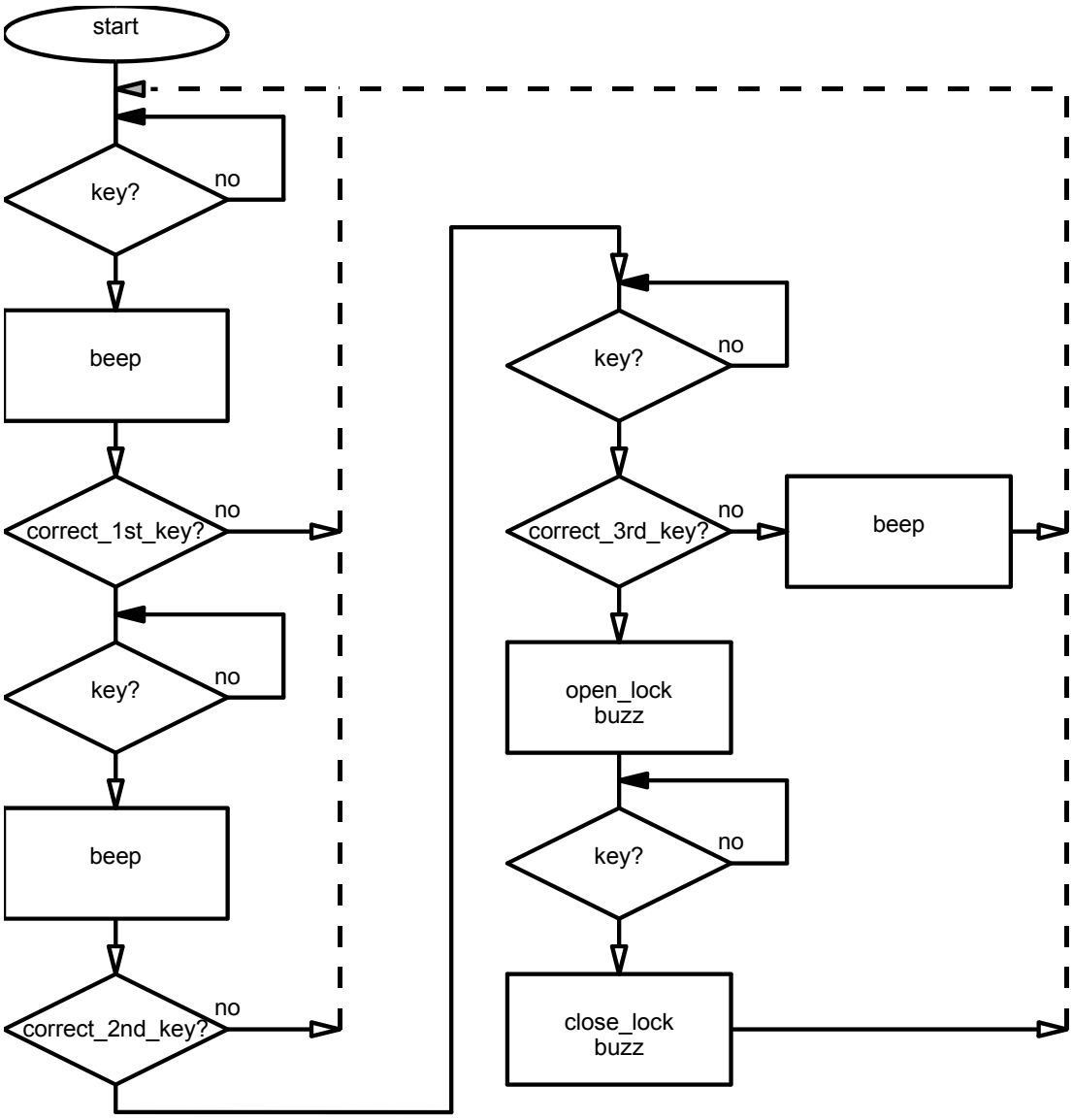
In this example a 10-key keypad controls an electric lock. To open the lock three keys must be entered in the correct sequence. Hence, there are four unique wait states.

One wait state waits for the 1st key to be entered. Upon a correct key entry, a transition will beep and take the machine to the second state. The second wait state waits for the 2nd key to be entered. Upon a correct key entry, a transition will beep and take the machine to the third state. The third wait state waits for the 3rd key to be entered. Upon a correct key entry, a transition will buzz, open the lock and take the machine to the fourth state. The fourth wait state waits for a key to be entered. Upon any key entry, a transition will buzz, again close the lock and take the machine back to the first state.

While the basic structure is quite similar to the previous problem, the four state garage door opener, there are three more transitions in this problem, not yet described. These transitions occur when a wrong key is entered in a particular wait state. The wrong key presses all return the state machine to the first state. This is the first instance where wait states have (in some instances) more than one entrance, and (in some instances) more than one exit. The second and third state have one entry and two exits. The first state has four entries and two exits.

One other new concept is shown in this example. A transition can originate in one state and return to anystate, including the originating one. When in the first state, entering a wrong key must cause an action, the beep. The destination state for every transition occassioned by a wrong key is the first state. The fact the first state is both the origin and the destination does not cause a problem for the state machine model.

The flow chart required to describe this problem is formidable even without accounting for Isostructure. The purpose of the algorithm is no longer obvious in this paradigm. It was no longer possible to present the whole program as a flow in a single column, so it was broken into two columns, each less than a page in length.



At this point, it is easy to see the old paradigm of flowcharting has failed to easily transmit the purpose of the algorythm to the viewer "in a glance," but rather requires significant study before the essence of the program becomes apparent. In short, the paradigm is more cumbersome than the problem, even without adding the additional requirement of Isostructure.

So, the graphical paradigm using state machine diagramming is the correct methodology to represent Isostructured problems. It has the advantages of explicit naming of wait states, which leads to the explicit creation of state information, and a terse syntax. It allows easier visualization of complex problems. The paradigm more closely resembles the problem and encourages Isostructure creation without representing flow based processing in Program Counter Capturing looping structures.

The next step toward Isostructured programming is a methodology to represent such programs in a human readable and writeable text language. IsoMax™ is that language.

Components

IsoMax™ is a text-based language allowing representation of Isostructure. The basic IsoMax™ words allow identification and definition of states and transitions. First, the key-words which define states will be covered.

```
STATE-MACHINE
ON-MACHINE
APPEND-STATE
```

States belong to a state machine. To gain concurrency, multiple state machines can be defined and run independently. In cases where machines are interrelated, communications between the machines allows interdependency as desired. So the highest level construct in IsoMax™ is the state machine. A state machine can be defined by using the key-word `STATE-MACHINE` followed by a name for the new state machine. It is used in the form `STATE-MACHINE <name of machine>`

Once a machine is defined, states may be appended to it. The key-words used in IsoMax™ to accomplish this are `ON-MACHINE` and `APPEND-STATE`. They are used in the form `ON-MACHINE <name of parent machine> APPEND-STATE <name of new state>`.

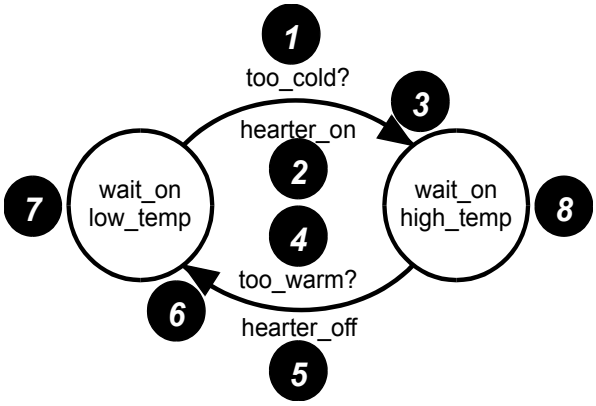
Now, the key-words which define transitions will be explained.

```
IN-STATE
CONDITION
CAUSES
THEN-STATE
TO-HAPPEN
```

Transitions connect related states in the state machine. To fully define a transition it is necessary to identify which state the transition will leave when valid, what condition is necessary to cause a valid transition, what action will be caused by the transition and which state will be entered after the action. These four components are sandwiched between the keywords `IN-STATE, CONDITION, CAUSES, THEN-STATE` and `TO-HAPPEN` to create a transition. They are used in the form: `IN-STATE <parent state name> CONDITION <boolean computation> CAUSES <compound action> THEN-STATE <next state> TO-HAPPEN.`

The state drawings made in the state machine paradigm can be directly translated into this ASCII paradigm. The previously used examples will be shown here (at the level of detail possible with the words so-far given). First the thermostat diagram will be shown

and then the code describing it listed. Numbers, similar to those used to show the corresponding elements between flow charts and state machine diagrams have been added to help give a visual correlation between graphical and textual elements:




```
STATE-MACHINE THERMOSTAT_A

     ON-MACHINE THERMOSTAT_A
          APPEND-STATE WAIT_ON_LOW_TEMP
          APPEND-STATE WAIT_ON_HIGH_TEMP

IN-STATE WAIT_ON_LOW_TEMP              ( 7 )
     CONDITION TOO_COLD?               ( 1 )
     CAUSES HEATER_ON                  ( 2 )
     THEN-STATE WAIT_ON_HIGH_TEMP      ( 3 )
     TO-HAPPEN

IN-STATE WAIT_ON_HIGH_TEMP             ( 8 )
     CONDITION TOO_WARM?               ( 4 )
     CAUSES HEATER_OFF                 ( 5 )
     THEN-STATE WAIT_ON_LOW_TEMP       ( 6 )
     TO-HAPPEN
```

Except for the meaning of the conditional `TOO_HOT?` and `TOO_COLD?` and the actions `HEATER_ON` and `HEATER_OFF`, the entire programming for the thermostat is shown above.

The first line, `STATE-MACHINE THERMOSTAT_A`, gives the new state machine a unique name. The second line, `ON-MACHINE THERMOSTAT_A`, identifies the selected state machine for addition of named states. (It is possible to define multiple state machines at the beginning of the program, then add their states to them later. Therefore, this line identifies which of the defined state machines is selected to receive new states.) The two lines `APPEND-STATE WAIT_ON_LOW_TEMP` and `APPEND-STATE WAIT_ON_HIGH_TEMP` give unique names to states appended to the selected state machine. The remaining two paragraphs describe the transitions.

These transition descriptions follow the form described in the previous section. Details of the transition are inserted between the keywords IN-STATE, CONDITION, CAUSES, THEN-STATE and TO-HAPPEN. The first transition described corresponds to the upper transition in the state machine diagram. It defines what is to happen in the state WAIT_ON_LOW_TEMP when the condition TOO_COLD? is in effect. In that case, the state machine causes HEATER_ON action to occur, followed by a change of state to the WAIT_ON_HIGH_TEMP state to happen. Look at the code written in the paragraph again. Once an understanding of the purpose is attained, the text reads like well written english:

```
IN-STATE WAIT_ON_LOW_TEMP
     CONDITION TOO_COLD?
     CAUSES HEATER_ON
     THEN-STATE WAIT_ON_HIGH_TEMP
     TO-HAPPEN
```

Written in linear form with the capitalization redone to resemble a more normal spoken sentence, the paragraph reads as: In-state WAIT_ON_LOW_TEMP condition TOO_COLD? causes HEATER_ON then-state WAIT_ON_HIGH_TEMP to-happen.

The last paragraph in the program follows the same format. It describes the lower transition on the graphic model. Applying the same modifications for the sake of illustration, it reads as: In-state WAIT_ON_HIGH_TEMP condition TOO_HOT? causes HEATER_OFF then-state WAIT_ON_LOW_TEMP to-happen.

The four yet undefined words: TOO_HOT?, TOO_COLD?, HEATER_ON and HEATER_OFF, are application specific, dealing with the actual hardware. They too must be defined. (Because of the method of single pass compilation, they must be defined before they are referenced.) However, since they are application specific and purely procedureal, without structure, they will be dealt with later. First the aspects of the language dealing with structure must be understood. Then we will return to detail the remaining "linear" portions of the programming elements (i.e. threads).

The rules of syntax in IsoMax™ are quite simple. The names of state machines and states may be composed of up to 31 non-whitespace characters. Transition clauses are not named. Names and keywords are separated by at least one space or a return and new line. The keywords STATE-MACHINE, ON-MACHINE, APPEND-STATE and IN-STATE are followed by names, which must be on the same line as the keyword. Other than these few rules, IsoMax™ is relatively syntax free. The example is repeated here with the minimum amount of "pretty" formatting:

```
STATE-MACHINE THERMOSTAT_A ON-MACHINE THERMOSTAT_A
APPEND-STATE WAIT_ON_LOW_TEMP APPEND-STATE WAIT_ON_HIGH_TEMP
IN-STATE WAIT_ON_LOW_TEMP CONDITION TOO_COLD?
CAUSES HEATER_ON THEN-STATE WAIT_ON_HIGH_TEMP TO-HAPPEN
IN-STATE WAIT_ON_HIGH_TEMP CONDITION TOO_WARM?
```

```
CAUSES HEATER_OFF THEN-STATE WAIT_ON_LOW_TEMP TO-HAPPEN
```

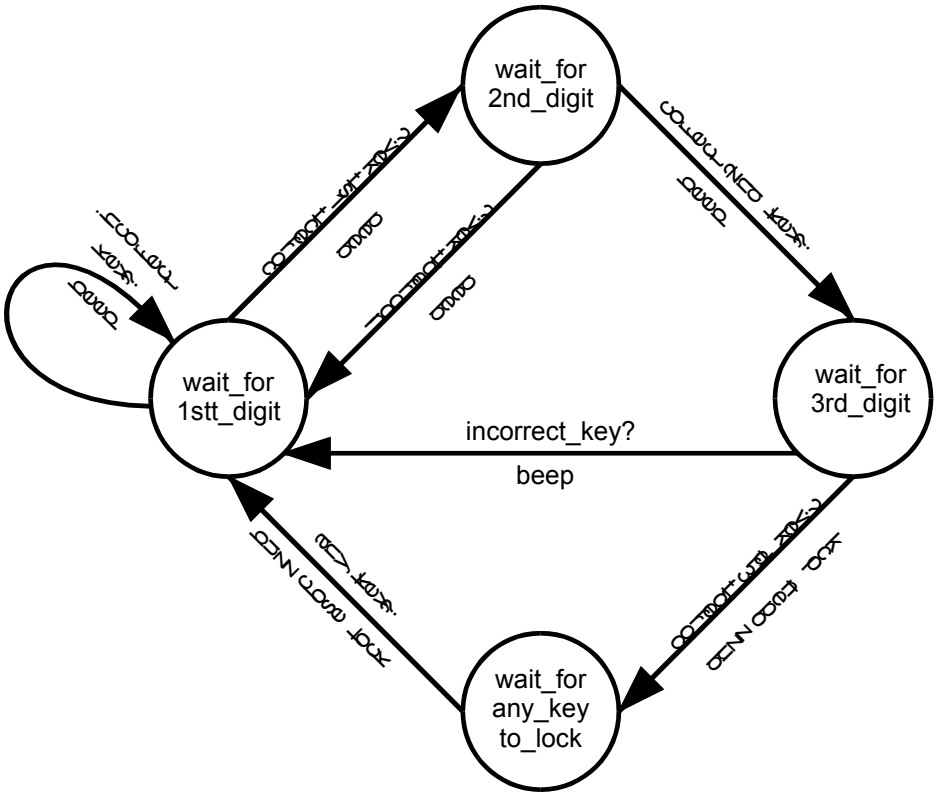
These above two listings are the same program and compiles the same code. Only the formatting has change.

Now that an example of the simplest useful state machine, one with two states and two transitions has been shown, a more complex example is in order. The garage door opener program follows as a further example:

```
_

STATE-MACHINE GARAGE_DOOR

     ON-MACHINE GARAGE_DOOR
          APPEND-STATE WAIT_TO_GO_DOWN
          APPEND-STATE WAIT_TO_STOP_DOWN
          APPEND-STATE WAIT_TO_GO_UP
          APPEND-STATE WAIT_TO_STOP_UP

IN-STATE WAIT_TO_GO_DOWN
     CONDITION OPERATOR_INPUT?
     CAUSES START_MOTOR_DOWN
     THEN-STATE WAIT_TO_STOP_DOWN
     TO-HAPPEN

IN-STATE WAIT_TO_STOP_DOWN
     CONDITION INPUT,LIMIT,OR_OVERCURRENT?
     CAUSES STOP_MOTOR
     THEN-STATE WAIT_TO_GO_UP
     TO-HAPPEN

IN-STATE WAIT_TO_GO_UP
     CONDITION OPERATOR_INPUT?
     CAUSES START_MOTOR_UP
     THEN-STATE WAIT_TO_STOP_UP
     TO-HAPPEN

IN-STATE WAIT_TO_STOP_UP
     CONDITION INPUT,LIMIT,OR_OVERCURRENT?
     CAUSES STOP_MOTOR
     THEN-STATE WAIT_TO_GO_DOWN
     TO-HAPPEN
```

Except for the meaning of the conditional and the actions the entire garage door program is shown above. Again, the state machine itself is named. This is necessary in order to activate the machine. Activating state machines will be discussed later. Next the state machine is selected as the machine on which transitions are to be added. Then the transitions are detailed one by one.

Now examine the 3-key sequence lock program:




```
STATE-MACHINE 3_KEY_LOCK

     ON-MACHINE 3_KEY_LOCK
          APPEND-STATE WAIT_FOR_1ST_DIGIT
          APPEND-STATE WAIT_FOR_2ND_DIGIT
          APPEND-STATE WAIT_FOR_3RD_DIGIT
          APPEND-STATE WAIT_FOR_ANY_KEY_TO_LOCK

IN-STATE WAIT_FOR_1ST_DIGIT
     CONDITION CORRECT_1ST_DIGIT?
     CAUSES BEEP
     THEN-STATE WAIT_FOR_2ND_DIGIT
     TO-HAPPEN

IN-STATE WAIT_FOR_1ST_DIGIT
     CONDITION INCORRECT_DIGIT?
     CAUSES BEEP
     THEN-STATE WAIT_FOR_1ST_DIGIT
     TO-HAPPEN

IN-STATE WAIT_FOR_2ND_DIGIT
     CONDITION CORRECT_2ND_DIGIT?
     CAUSES BEEP
```

```
     THEN-STATE WAIT_FOR_3RD_DIGIT
     TO-HAPPEN

IN-STATE WAIT_FOR_2ND_DIGIT
     CONDITION INCORRECT_DIGIT?
     CAUSES BEEP
     THEN-STATE WAIT_FOR_1ST_DIGIT
     TO-HAPPEN

IN-STATE WAIT_FOR_3RD_DIGIT
     CONDITION CORRECT_3RD_DIGIT?
     CAUSES BUZZ OPEN_LOCK
     THEN-STATE WAIT_FOR_ANY_KEY_TO_LOCK
     TO-HAPPEN

IN-STATE WAIT_FOR_3RD_DIGIT
     CONDITION INCORRECT_DIGIT?
     CAUSES BEEP
     THEN-STATE WAIT_FOR_1ST_DIGIT
     TO-HAPPEN

IN-STATE WAIT_FOR_ANY_KEY_TO_LOCK
     CONDITION CORRECT_1ST_DIGIT?
     CAUSES BUZZ CLOSE_LOCK
     THEN-STATE WAIT_FOR_1ST_DIGIT
     TO-HAPPEN
```

Other than having more transitions, the program has no new elements. Once Isostructure is understood, it is easily written with IsoMax™ with great ease and regularity.

To review: An state machine program is composed of three distinct parts:

The name of the state machine.
The names of the states which belong to the state machine. These correspond to the bubbles in a state machine diagram.
The "transition clauses" which define the conditions and actions of the state machine. These correspond to the arrows linking the states together and indicate
the state in which the arrow originates
the condition which must be true for the state machine to take the transition
the action to take when the condition is true and
the state which is to become active after the action.

System

The previous examples explained how Isostructure was implemented in IsoMax™. They were not complete examples, however. Application specific words were left undefined.

In order to show a full system program and advance into the higher concepts of how the IsoMax™ operating system works, a few of these functions must be described.

Special operators are provided in IsoMax™ to accomodate the simplest forms of Input/Output, these being single digital lines. These special operators in IsoMax™ are called trinary operators. They allow manipulation or testing of bits. Often a single port line will represent an external real-world condition. Trinary operators can deal with individual bits.

In the case of a thermostat, for instance, a contact closure could represent the condition of the temperature being too low. When the bit is in a given state, either high or low, the temperature is known to be too low. Similarly, another bit may represent the condition too hot. These are examples of single bit inputs.

Outputs may also be single port lines. A port being in a given state, high or low, may control a heater being on or off. In the case of the thermostat, the port line may activate a transistor, which operates a mechanical or solid state relay which then applies powers to the actual heater coils.

The difficulty dealing with single input/output points in microcomputers is, lines never occur as single points, but instead are grouped in I/O ports. It's impossible to write one bit in an I/O port without writting them all. So in most control languages, to change a single bit in an I/O port, the output port must be read, the single bit modified, and the output port written with the modified value. In IsoMax™ output trinary operators do read-modify-write operations and, therefore, make dealing with ports easier. Input trinary operators of a similar nature make reading selected input bits out of a full port easier, too. These operators are called trinary operators, because they take three values and create a single named action.
Programming consists in defining actions for new words, named by the programmer. To begin a new definition, the keyword `DEFINE` is used, followed by a unique name. The same rules apply as do for state machine and state names. Names may be composed of up to 31 non-whitespace characters. The new name must follow the defining word on the same line.

Once a new word is defined, its purpose must be qualified. In the case of the trinary operators now under discussion, the three defining parameters and the type of operator must be specified. Input trinary operators need three parameters: 1) a mask telling which bits in the input port are active, 2) a mask telling which state the active bits must be in, and 3) the address of the I/O port. The keywords which procede the parameters are, in order: 1) `TEST-MASK`, 2) `DATA-MASK` and 3) `AT-ADDRESS`. Finally, the keyword `FOR-INPUT` finishes the defining process, identifying which trinary operator is in effect.

The following lines might be used in the thermostat program. A 68HC11 target processor is assumed which has Port A at $B000. On the 68HC11 PA0-PA2 are inputs. PA4-PA6 are outputs. (PA3 and PA7 are programmable as inputs or outputs, but default to being

inputs, unless specifically programmed otherwise.) In the case of the thermostat, if historesis is built into the mechanical thermostat switch, only one input line is needed.

PA0 was selected for the input signal. For the condition TOO-COLD? a test mask of 01 indicates only the least significant bit in the port, corresponding to PA0, is active in testing. A data mask of 01 inidcates the least significant bit in the port is tested for being set when the returned boolean conditional is true.

For the condition TOO-HOT? a test mask of 01 indicates only the least significant bit in the port, corresponding to PA0, is active in testing. A data mask of 00 inidcates the least significant bit in the port is tested for being clear when the returned boolean conditional is true. (In this case, TOO-HOT? is the logical oppositie of the TOO-COLD? condition.)

Putting the keywords and parameters together produces the following lines of IsoMax™ code. Before entering hexidecimal numbers, the keyword HEX invokes the use of the hexidecimal number system. This remains in effect until it is change by a later command. The numbering system can be returned to decimal using the keyword DECIMAL:

```
HEX
DEFINE TOO-COLD? TEST-MASK 01 DATA-MASK 01 AT-ADDRESS B000
FOR-INPUT
DEFINE TOO-HOT?  TEST-MASK 01 DATA-MASK 00 AT-ADDRESS B000
FOR-INPUT
DECIMAL
```

Output trinary operators also need three parameters. In this instance, using the trinary operation mode of setting and clearing bits would be convenient. This mode requires: 1) a mask telling which bits in the output port are to be set, 2) a mask telling which bits in the output port are to be cleared, and 3) the address of the I/O port. The keywords which procede the parameters are, in order: 1) SET-MASK, 2) CLR-MASK and 3) AT-ADDRESS. Finally, the keyword FOR-OUTPUT finishes the defining process, identifying which trinary operator is in effect.

A single output port line is needed to turn the heater on and off. The act of turning the heater on is unique and different from turning the heater off, however. Two actions need to be defined, therefore, even though only one I/O line is involved. PA4 was selected for the heater control signal.

When PA4 is high, or set, the heater is turned on. To make PA4 high, requires PA4 to be set, without changing any other bit of the port. Therefore, a set mask of 10 indicates the least significant bit in the high order nibble of the port, corresponding to PA4, is to be set. All other bits are to be left alone without being set. A clear mask of 00 inidcates no other bits of the port are to be cleared.

When PA4 is low, or clear, the heater is turned off. To make PA4 low, requires PA4 to be cleared, without changing any other bit of the port. Therefore, a set mask of 00 indicates

no other bits of the port are to be set. A clear mask of 10 inidcates the least significant bit in the high order nibble of the port, corresponding to PA4, is to be cleared. All other bits are to be left alone without being cleared.

Putting the keywords and parameters together produces the following lines of IsoMax™ code:

```
HEX
DEFINE HEATER-ON  SET-MASK 10 CLR-MASK 00 AT-ADDRESS B000
FOR-OUTPUT
DEFINE HEATER-OFF SET-MASK 00 CLR-MASK 10 AT-ADDRESS B000
FOR-OUTPUT
DECIMAL
```

With the Isostructure given, and the trinary operators, most digital control applications can be written. Only a handful of system words need to be covered to allow programming at a system level, now.

```
MACHINE-CHAIN ALL-TASKS
   THERMOSTAT_A
END-MACHINE-CHAIN

DECIMAL
: MAIN
   SEI
   WAIT_ON_LOW_TEMP SET-STATE
   HEATER-ON
   EVERY 20000 CYCLES SCHEDULE-RUNS ALL-TASKS
   CLI ;

XXXX AUTOSTART MAIN
```

Procedures

Up to this point, only the structure of IsoMax™ has been highlighted. Procedures have been limited to a single word. Conditionals in transistions have been defined as a single word. Actions have been defined as a single word. The examples were limited to conditions and actions which could be represented as a simple boolean. While a great number of real time applications require nothing greater, many other applications inputs and outputs are more complex.

To illustrate, reconsider the simple thermostat. The conditions `TOO-COLD?` and `TOO-HOT?` can be represented as a single binary digit, or as a simple boolean. In the physical world, to allow proper hysteresis, this would be constructed as thermostat contact closures on a bimetalic strip with a glass tube containing a bead of mercury. When the

bimetalic strip cools and contracts enough to tip the glass tube one way, the bead of mercury moves from the one end of the tube (hot position) to the other (cold position). In so moving, it weights down the new end (cold position) which resists being tilted back. When the bimetalic strip heats and expands enough to tip the glass tube the other way, the bead of mercury moves from the one end of the tube (cold position) to the other (hot position). In so moving, it weights down the new end (hot position) which resists being tilted back.

This arrangement takes a number of control factors out of the realm of control for the microprocessor. The temperature setting, and the range of hysteresis is set mechanically. A better approach would be to use an analog temperature sensor, read the temperature with the microprocessor, and then set point and range (or low point and high point) under program control.

The reading of an A/D converter and comparison to a given value, are not as simple as testing a boolean value. Some procedureal processing is required before a decision can be reached, or put another way, a boolean value calculated for evaluation.

To simplify the examples up to this point, the trinary operators where introduced, which defined simple boolean inputs and outputs as single named words. Procedures can be used instead of single named words.

For instance, in the thermostat example, an A/D reading can be taken from A/D registers if the A/D control register has been set up. That A/D value can then be compared to a preset limit. The result of that comparison, a boolean, can be used instead of the trinary operator as the object of the conditional. Disregarding the set up of the A/D control register, the modified code for the state machine would look like this:

```
STATE-MACHINE THERMOSTAT_A

     ON-MACHINE THERMOSTAT_A
          APPEND-STATE WAIT_ON_LOW_TEMP
          APPEND-STATE WAIT_ON_HIGH_TEMP

IN-STATE WAIT_ON_LOW_TEMP
     CONDITION B031 C@ LOW-LIMIT @ U< ( TOO_COLD?
     CAUSES HEATER_ON
     THEN-STATE WAIT_ON_HIGH_TEMP
     TO-HAPPEN

IN-STATE WAIT_ON_HIGH_TEMP
     CONDITION HIGH-LIMIT @ B031 C@ U< ( TOO_WARM?
     CAUSES HEATER_OFF
     THEN-STATE WAIT_ON_LOW_TEMP
     TO-HAPPEN
```

It is not necessary to insert the entire procedure into the area between the CONDITION and CAUSES portion of the transition code. Nor is it likely to be desireable to do so. (Factoring and information hiding are popular descriptions of the methodology of separating code into "parcel" size components. This factoring makes code easier to test and maintian.) A better approach would be to define and name the procedure.

```
DEFINE TOO-COLD?
  PROC B031 C@ LOW-LIMIT @ U< END-PROC
DEFINE TOO-HOT?
  PROC HIGH-LIMIT @ B031 C@ U< END-PROC

STATE-MACHINE THERMOSTAT_A

     ON-MACHINE THERMOSTAT_A
          APPEND-STATE WAIT_ON_LOW_TEMP
          APPEND-STATE WAIT_ON_HIGH_TEMP

IN-STATE WAIT_ON_LOW_TEMP
     CONDITION TOO_COLD?
     CAUSES HEATER_ON
     THEN-STATE WAIT_ON_HIGH_TEMP
     TO-HAPPEN

IN-STATE WAIT_ON_HIGH_TEMP
     CONDITION TOO_WARM?
     CAUSES HEATER_OFF
     THEN-STATE WAIT_ON_LOW_TEMP
     TO-HAPPEN
```

Trinaries

```
DEFINE TOO-COLD? TEST-MASK 01 DATA-MASK 01 AT-ADDRESS B000
FOR-INPUT
DEFINE TOO-HOT?  TEST-MASK 02 DATA-MASK 02 AT-ADDRESS B000
FOR-INPUT

DEFINE HEATER-ON  SET-MASK 10 CLR-MASK 00 AT-ADDRESS B000
FOR-OUTPUT
DEFINE HEATER-OFF SET-MASK 00 CLR-MASK 10 AT-ADDRESS B000
FOR-OUTPUT

DATA-MASK
```

```
TEST-MASK

SET-MASK
CLR-MASK

AND-MASK
XOR-MASK
```

Counter/Timers

```
DEFINE 1SEC  COUNTDOWN-TIMER

   1000 TIMER-INIT 1SEC
```

Debugging

```
   ON-MACHINE PULSE-RED
        APPEND-STATE RED-ON     WITH-VALUE 00 AT-ADDRESS
7FFF  AS-TAG
        APPEND-STATE RED-OFF    WITH-VALUE FF AT-ADDRESS
7FFF  AS-TAG
```