PROGRAMMING

Under construction…

IsoMax is a programming language based on Finite State Machine (FSM) concepts applied to software, with a procedural language based derived from Forth underneath it. The closest description to the FSM construction type is a “One-Hot” Mealy type of Timer Augmented Finite State Machines. More on these concepts will come later.

QUICK OVERVIEW

What is IsoMax™? IsoMax™ is a real time operating system / language.

How do you program in IsoMax™? You create state machines that can run in a virtually parallel architecture.

What do you have to write to make a state machine in IsoMax™? You give a machine a name, and then tell the system that’s the name you want to work on. You append any number of states to the machine. You describe any number of transitions between states. Then you test the machine and when satisfied, install it into the machine chain.

Step	**Programming Action**	**Syntax**
1	Name a state machine	`MACHINE <name>`
2	Select this state	`ON-MACHINE <name>`
3	Name any states appended on the machine	`APPEND-STATE <name>`

4	Describe transitions from states to states	`IN-STATE`
		`  <state>`
		`CONDITION`
		`  <Boolean>`
		`CAUSES`
		`  <action>`
		`THEN-STATE`
		`  <state>`
		`TO-HAPPEN`

5	Test and Install	{as required}

What is a transition? A transition is how a state machine changes states.

What’s in a transition? A transition has four components; 1) which state it starts in, 2) the condition necessary to leave, 3) the action to take when the condition comes true, and 4) the state to go to next time. Why are transitions so verbose? The structure makes the transitions easy to read in human language. The constructs `IN-STATE, CONDITION, CAUSES, THEN-STATE` and `TO-HAPPEN` are like the five brackets around a table of four things.

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

<from state>	<Boolean>	<action>	<to state>

In a transition description the constructs `IN-STATE`, `CONDITION`, `CAUSES`, `THEN-STATE` and `TO-HAPPEN` are always there (with some possible options to be set out later). The "meat slices" between the "slices of bread" are the hearty stuffing of the description. You will fill in those portions to your own needs and liking. The language provides "the bread" (with only a few options to be discussed later).

So here you have learned a bit of the syntax of IsoMax™. Machines are defined, states appended. The transitions are laid out in a pattern, with certain words surrounding others. Procedural parts are inserted in the transitions between the standard clauses. The syntax is very loose compared to some languages. What is important is the order or sequence these words come in. Whether they occur on one line or many lines, with one space or many spaces between them doesn't matter. Only the order is important.

THREE MACHINES

Now let's take a first step at exploring IsoMax™ the language. We'll explore the language with what we've just tested earlier, the LED words. We'll add some machines that will use the LED's as outputs, so we can visually "see" how we're coming along.

REDTRIGGER

First let's make a very simple machine. Since it is so short, at least in V0.3 and later, it's presented first without detailed explanation, entered and tested. Then we will explain the language to create the machine step by step

```
( THESE GRAY'D TEXT LINES ARE PATCHES FOR V0.2 UPDATE TO V0.3
( IF YOU"VE GOT V0.2 JUST ENTER GRAY'D VERBATUM.
( IF YOU'VE GOT V0.3, IGNORE, ALREADY PUT IN THE LANGUAGE

HEX
: OFF?
  1 =
  IF
    2DUP 3 + @ SWAP FFFF XOR AND OVER 3 + !
    2DUP 2 + @ SWAP FFFF XOR AND OVER 2 + !
    1 + @ AND 0=
  ELSE
    SWAP DROP DUP @ FCFE AND OVER ! @ FF7F AND 0=
  THEN
;
DECIMAL
```

```
MACHINE REDTRIGGER ON-MACHINE REDTRIGGER APPEND-STATE RT
IN-STATE RT CONDITION PA7 OFF? CAUSES REDLED ON THEN-STATE RT TO-HAPPEN

RT SET-STATE ( INSTALL REDTRIGGER
EVERY 50000 CYCLES SCHEDULE-RUNS REDTRIGGER
```

There you have it, a complete real time program in two lines of IsoMax™, and one additional line to install it. A useful virtual machine is made here with one state and one transition.

This virtual machine acts like a non-retriggerable one-shot made in hardware. If PA7 goes low briefly, the red LED turns on and stays on even if PA7 then changes. PA7 normally has a pull up resistor that will keep it “on”, or “high” if nothing is attached. So attaching push button from PA7 to ground, or even hooking a jumper test lead to ground and pushing the other end into contact with the wire lead in PA7, will cause PA7 to go “off” or “low”, and the `REDLED` will come on.

(In these examples, any port line that can be an input could be used. PA7 here, PB7 and PB6 later, were chosen because they are at the bottom of J1 and the easiest for you to access.)

Now if you want, type these lines shown above in. (If you are reading this manual electronically, you should be able to highlight the text on screen and copy the text to the clipboard with Cntl-C. Then you may be able to paste into your terminal program. On MaxTerm, the command to down load the clipboard is Alt-V. On other windows programs it might be Cntl-V.)

Odds are your red LED is already on. When the IsoPod™ powers up, it’s designed to have the LED’s on, unless programmed otherwise by the user. So to be useful we must reset this one-shot. Enter:

```
REDLED OFF
```

Now install the `REDTRIGGER` by installing it in the (now empty) machine chain.

```
RT SET-STATE ( INSTALL REDTRIGGER
EVERY 50000 CYCLES SCHEDULE-RUNS REDTRIGGER
```

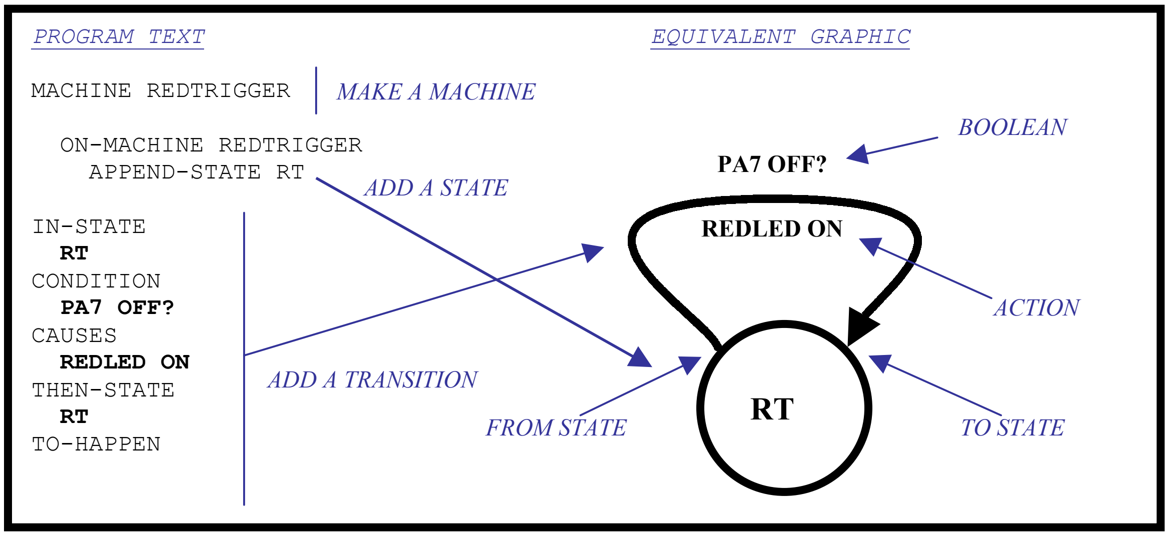
Ground PA7 with a wire or press the push button, and see the red LED come on. Remove the ground or release the push button. The red LED does not go back off. The program is still running, even though all visible changes end at that point. To see that, we’ll need to manually reset the LED off so we can see something happen again. Enter.

```
REDLED OFF
```

If we ground PA7 again, the red LED will come back on, so even though we are still fully interactive with the IsoPod™, the `REDTRIGGER` machine is running in the background.

Now let's go back through the code, step-by-step. We'll take it nice and easy. We'll take the time explain the concepts of this new language we skipped over previously.

Here in this box, the code for `REDTRIGGER` "pretty printed" so you can see how the elements of the program relate to a state machine diagram. Usually you start to learn a language by learning the syntax, or how and where elements of the program must be placed. The syntax of the IsoMax™ language is very loose. Almost anything can go on any line with any amount of white space between them as long as the sequence remains the same. So in the pretty printing, most things are put on a separate line and have spaces in front of them just to make the relationships easy to see. Beyond the basic language syntax, a few words have a further syntax associated to them. They must have new names on the same line as them. In this example, `MACHINE`, `ON-MACHINE` and `APPEND-STATE` require a name following. You will see that they do. More on syntax will come later.



```
MACHINE REDTRIGGER

  ON-MACHINE REDTRIGGER
    APPEND-STATE RT

IN-STATE
  RT
CONDITION
  PA7 OFF?
CAUSES
  REDLED ON
THEN-STATE
  RT
TO-HAPPEN
```

In this example, the first program line, we'll tell IsoMax™ we're making a new virtual machine, named `REDTRIGGER`. (Any group of characters without a space or a backspace or return will do for a name. You can be very creative. Use up to 32 characters. Here the syntax is `MACHINE` followed by the chosen name.)

```
MACHINE REDTRIGGER
```

That's it. We now have a new machine. This particular new machine is named `REDTRIGGER`. It doesn't do anything yet, but it is part of the language, a piece of our program.

For our second program line, we'll identify `REDTRIGGER` as the machine we want to append things to. The syntax to do this is to say `ON-MACHINE` and the name of the machine we want to work on, which we named `REDTRIGGER` so the second program line looks like this:

```
  ON-MACHINE REDTRIGGER
```

(Right now, we only have one machine installed. We could have skipped this second line. Since there could be several machines already in the IsoPod™ at the moment, it is good policy to be explicit. Always use this line before appending states. When you have several machines defined, and you want to add a state or transition to one of them, you will need that line to pick the machine being appended to. Otherwise, the new state or transition will be appended to the last machine worked on.)

All right. We add the machine to the language. We have told the language the name of the machine to add states to. Now we'll add a state with a name. The syntax to do this is to say APPEND-STATE followed by another made-up name of our own. Here we add one state `RT` like this:

```
    APPEND-STATE RT
```

States are the fundamental parts of our virtual machine. States help us factor our program down into the important parts. A state is a place where the computer's outputs are stable, or static. Said another way, a state is place where the computer waits. Since all real time programs have places where they wait, we can use the waits to allow other programs to have other processes. There is really nothing for a computer to do while its outputs are stable, except to check if it is time to change the outputs.

(One of the reasons IsoMax™ can do virtually parallel processing, is it never allows the computer to waste time in a wait, no backwards branches allowed. It allows a check for the need to leave the state once per scheduled time, per machine.)

To review, we've designed a machine and a sub component state. Now we can set up something like a loop, or jump, where we go out from the static state when required to do some processing and come back again to a static wait state.

The rules for changing states along with the actions to do if the rule is met are called transitions. A transition contains the name of the state the rule applies to, the rules called the condition, what to do called the action, and "where to go" to get into another state. (We have only one state in this example, so the last part is easy. There is no choice. We go back into the same state. In machines with more than one state, it is obviously important to have this final piece.)

There's really no point in have a state in a machine without a transition into or out of it. If there is no transition into or out of a state, it is like designing a wait that cannot start, cannot end, and cannot do anything else either.

On the other hand, a state that has no transition into it, but does have one out of it, might be an "initial state" or a "beginning state". A state that has a transition into it, but doesn't have one out of it, might be a "final state" or an "ending state". However, most states will have at least one (or more) transition entering the state and one (or more) transition leaving the state. In our example, we have one transition that leaves the state, and one that comes into the state. It just happens to be the same one.

Together a condition and action makes up a transition, and transitions go from one specific state to another specific state. So there are four pieces necessary to describe a transition; 1) The state the machine starts in. 2) the condition to leave that state 3) the action taken between states and 4) the new state the machine goes to.

Looking at the text box with the graphic in it, we can see the transitions four elements clearly labeled. In the text version, these four elements are printed in bold. In the equivalent graphic they are labeled as "FROM STATE", "BOOLEAN", "ACTION" and "TO STATE".

The "FROM STATE" is `RT`. The "BOOLEAN" is a simple phrase checking I/O `PA7 OFF?`. The "ACTION" is `REDLED ON`. The "TO STATE" is again `RT`.

So to complete our state machine program, we must define the transition we need. The syntax to make a transition, then, is to fill in the blanks between this form: `IN-STATE <name> CONDITION <Boolean> CAUSES <action> THEN-STATE <name> TO-HAPPEN`.

Whether the transition is written on one line as it was at first:

```
IN-STATE RT CONDITION PA7 OFF? CAUSES REDLED ON THEN-STATE RT TO-HAPPEN
```

Or pretty printed on several lines as it was in the text box:

```
IN-STATE
  RT
CONDITION
  PA7 OFF?
CAUSES
  REDLED ON
THEN-STATE
  RT
TO-HAPPEN
```

The effect is the same. The five boarder words are there, and the four user supplied states, condition and action are in the same order and either way do the same thing.

After the transition is added to the program, the program can be tested and installed as shown above.

State machine diagrams (the graphic above being an example) are nothing new. They are widely used to design hardware. They come with a few minor style variations, mostly related to how the outputs are done. But they are all very similar.

While FSM diagrams are also widely known in programming as an abstract computational element, there are few instances where they are used to design software. Usually, the tools for writing software in state machines are very hard to follow. The programming style doesn't seem to resemble the state machine design, and is often a slow, table-driven "read, process all inputs, computation and output" scheme.

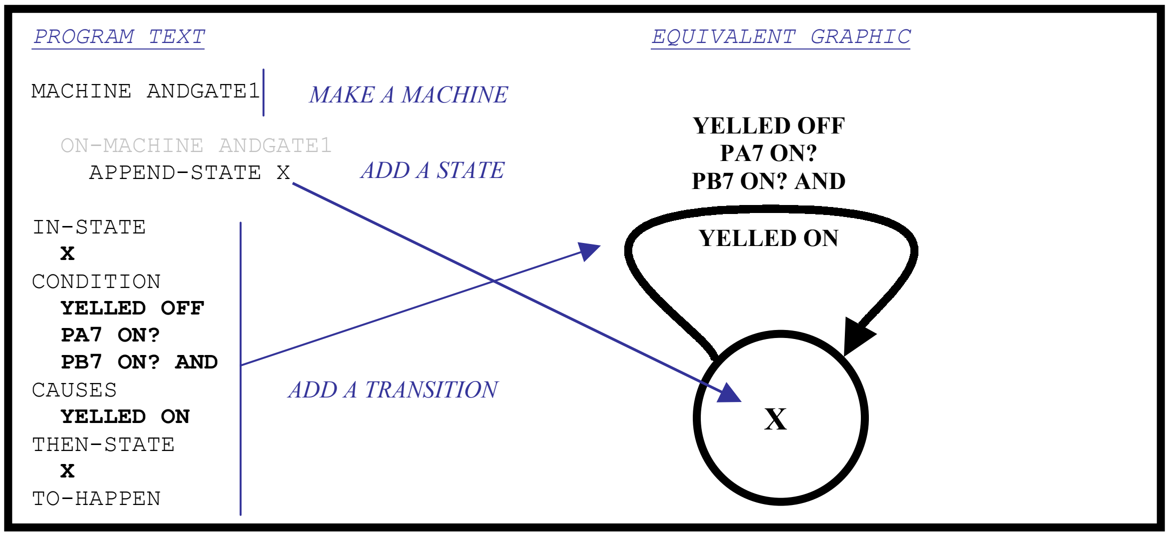
IsoMax™ technology has overcome this barrier, and gives you the ability to design software that looks “like” hardware and runs “like” hardware (not quite as fast of course, but in the style, or thought process, or “paradigm” of hardware) and is extremely efficient. The Virtually Parallel Machine Architecture lets you design many little, hardware-like, machines, rather than one megalith software program that lumbers through layer after layer of if-then statements. (You might want to refer to the IsoMax Reference Manual to understand the language and its origins.)

ANDGATE1

Let’s do another quick little machine and install both machines so you can see them running concurrently.

```
( THESE GREY'D TEXT LINES ARE PATCHES FOR V0.2 UPDATE TO V0.3

HEX
: ON?
  1 =
  IF
    2DUP 3 + @ SWAP FFFF XOR AND OVER 3 + !
    2DUP 2 + @ SWAP FFFF XOR AND OVER 2 + !
    1 + @ AND
  ELSE
    SWAP DROP DUP @ FCFE AND OVER ! @ FF7F AND 0= NOT
  THEN
;
DECIMAL

MACHINE ANDGATE1 ON-MACHINE ANDGATE1 APPEND-STATE X
IN-STATE X CONDITION YELLED OFF PA7 ON? PB7 ON? AND CAUSES YELLED ON THEN-STATE
X TO-HAPPEN

X SET-STATE ( INSTALL ANDGATE1
MACHINE-CHAIN CHN1 REDTRIGGER ANDGATE1 END-MACHINE-CHAIN
EVERY 50000 CYCLES SCHEDULE-RUNS CHN1
```

There you have it, another complete real time program in three lines of IsoMax™, and one additional line to install it. A useful virtual machine is made here with one state and one transition. This virtual machine acts (almost) like an AND gate made in hardware. Both PA7 and PB7 must be on, or high, to allow the yellow LED to remain on (most of the time). So by attaching push buttons to PA7 and PB7 simulating micro switches this little program could be used like an interlock system detecting “cover closed”.



```
MACHINE ANDGATE1

  ON-MACHINE ANDGATE1
    APPEND-STATE X

IN-STATE
  X
CONDITION
  YELLED OFF
  PA7 ON?
  PB7 ON? AND
CAUSES
  YELLED ON
THEN-STATE
  X
TO-HAPPEN
```

(Now it is worth mentioning, the example is a bit contrived. When you try to make a state machine too simple, you wind up stretching things you shouldn't. This example could have acted exactly like an AND gate if two transitions were used, rather than just one. Instead, a "trick" was used to turn the LED off every time in the condition, then turn it on only when the condition was true. So a noise spike is generated a real "and" gate doesn't have. The trick made the machine simpler, it has half the transitions, but it is less functional. Later we'll revisit this machine in detail to improve it.)

Notice both machines share an input, but are using the opposite sense on that input. `ANDGATE1` looks for PA7 to be ON, or HIGH. The internal pull up will normally make PA7 high, as long as it is programmed for a pull up and nothing external pulls it down.

Grounding PA7 enables `REDTRIGGER`'s condition, and inhibits `ANDGATE1`'s condition. Yet the two machines coexist peacefully on the same processor, even sharing the same inputs in different ways.

To see these machines running enter the new code, if you are still running `REDTRIGGER`, reset (toggle the DTR line on the terminal, for instance, Alt-T twice in MaxTerm or cycle power) and download the whole of both programs.

Initialize `REDTRIGGER` for action by turning `REDLED OFF` as before. Grounding PA7 now causes the same result for `REDTRIGGER`, the red LED goes on, but the opposite effect for the yellow LED, which goes off while PA7 is grounded. Releasing PA7 turns the yellow LED back on, but the red LED remains on.

Again, initialize `REDTRIGGER` by turning `REDLED OFF`. Now ground PB7. This has no effect on the red LED, but turns off the yellow LED while grounded. Grounding both PA7 and PB7 at the same time also turns off the yellow LED, and turns on the red LED if not yet set.

Notice how the tightly the two machines are intertwined. Perhaps you can imagine how very simple machines with combinatory logic and sharing inputs and feeding back outputs can quickly start showing some complex behaviors.

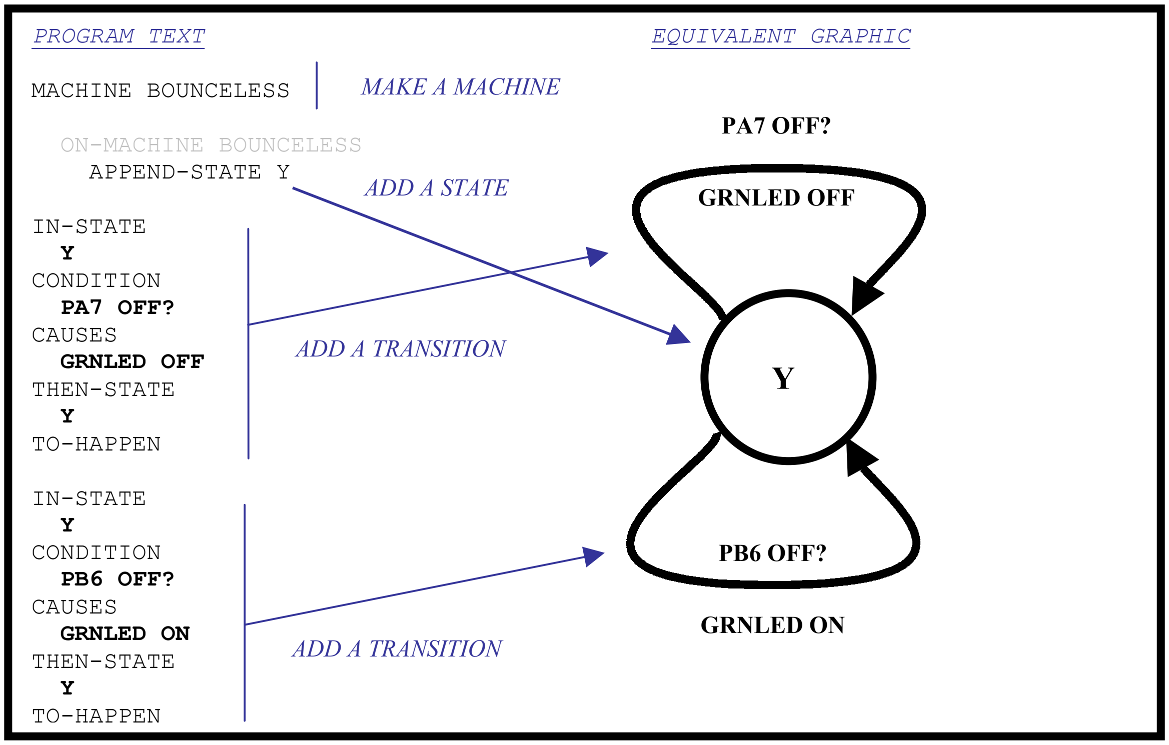
BOUNCELESS

We have another quick example of a little more complex machine, one with one state and two transitions.

```
MACHINE BOUNCELESS ON-MACHINE BOUNCELESS APPEND-STATE Y
IN-STATE Y CONDITION PA7 OFF? CAUSES GRNLED OFF THEN-STATE Y TO-HAPPEN
IN-STATE Y CONDITION PB6 OFF? CAUSES GRNLED ON THEN-STATE Y TO-HAPPEN

Y SET-STATE ( INSTALL BOUNCELESS

MACHINE-CHAIN 3EASY
REDTRIGGER
ANDGATE
BOUNCELESS
END-MACHINE-CHAIN

EVERY 50000 CYCLES SCHEDULE-RUNS 3EASY
```

There you have yet another complete design, initialization and installation of a virtual machine in four lines of IsoMax™ code.

Another name for the machine in this program is “a bounceless switch”. Bounceless switches filter out any noise on their input buttons, and give crisp, one-edge output signals. By attaching push buttons to PA7 and PB6 the green LED can be toggled from on to off with the press of the PA7 button, or off to on with the press of the PB6. The PA7 button acts as a reset switch, and the PB6 acts as a set switch



```
MACHINE BOUNCELESS

  ON-MACHINE BOUNCELESS
    APPEND-STATE Y

IN-STATE
  Y
CONDITION
  PA7 OFF?
CAUSES
  GRNLED OFF
THEN-STATE
  Y
TO-HAPPEN

IN-STATE
  Y
CONDITION
  PB6 OFF?
CAUSES
  GRNLED ON
THEN-STATE
  Y
TO-HAPPEN
```

You can see here, in IsoMax™, you can simulate hardware machines and circuits, with just a few lines of code. Here we created one machine, gave it one state, and appended two transitions to that state. Then we installed the finished machine along with the two previous machines. All run in the background, freeing us to program more virtual machines that can also run in parallel, or interactively monitor existing machines from the foreground.

Notice all three virtual hardware circuits are installed at the same time, they operate virtually in parallel, and the IsoPod™ is still not visibly taxed by having these machines run in parallel.

SYNTAX AND FORMATTING

Let's talk a second about pretty printing, or pretty formatting. To go a bit into syntax again, you'll need to remember the following. Everything in IsoMax™ is a word or a number. Words and numbers are separated spaces (or returns).

Some words have a little syntax of their own. The most common cases for such words are those that require a name to follow them. When you add a new name, you can use any combinations of characters or letters except (obviously) spaces and backspaces, and carriage returns. So, when it comes to pretty formatting, you can put as much on one line as will fit (up to 80 characters). Or you can put as little on one line as you wish, as long as you keep your words whole. However, some words will require a name to follow them, so those names will have to be on the same line.

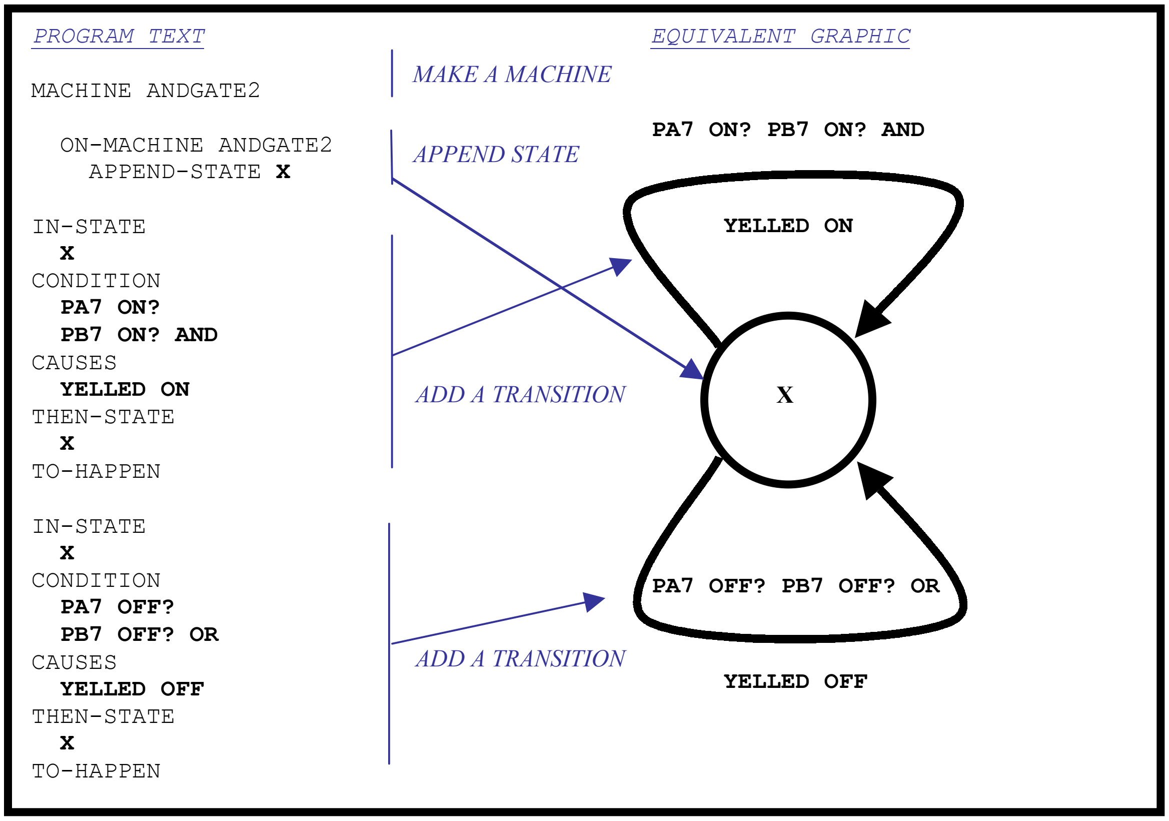
In the examples you will see white space (blanks) used to add some formatting to the source text. `MACHINE` starts at the left, and is followed by the name of the new machine being added to the language. `ON-MACHNE` is indented right by two spaces. `APPEND-STATE X` is indented two additional spaces. This is the suggested, but not mandatory, offset to achieve pretty formatting. Use two spaces to indent for levels. The transitions are similarly laid out, where the required words are positioned at the left, and the user programming is stepped in two spaces.

MULTIPLE STATES/MULTIPLE TRANSITIONS

Before we leave the previous “Three Machines”, let’s review the AND machine again, since it had a little trick in it to keep it simple, just one state and one transition. The trick does simplify things, but goes too far, and causes a glitch in the output. To make an AND gate which is just like the hardware AND we need at least two transitions. The previous example, `BOUNCELESS` was the first state machine with more than one transition. We’ll follow this precedent and redo `ANDGATE2` with two transitions.

ANDGATE2

```
( THESE GREY'D TEXT LINES ARE PATCHES FOR V0.2 UPDATE TO V0.3
( ASSUME ON? ALREADY DEFINED AS IN OTHER PROGRAM

MACHINE ANDGATE2
  ON-MACHINE ANDGATE2
    APPEND-STATE X

IN-STATE
  X
CONDITION
  PA7 ON?
  PB7 ON? AND
CAUSES
  YELLED ON
THEN-STATE
  X
TO-HAPPEN

IN-STATE
  X
CONDITION
  PA7 OFF?
  PB7 OFF? OR
CAUSES
  YELLED OFF
THEN-STATE
  X
TO-HAPPEN

X SET-STATE ( INSTALL ANDGATE2
EVERY 50000 CYCLES SCHEDULE-RUNS ANDGATE2
```



```
PROGRAM TEXT

MACHINE ANDGATE2

  ON-MACHINE ANDGATE2
    APPEND-STATE X

IN-STATE
  X
CONDITION
  PA7 ON?
  PB7 ON? AND
CAUSES
  YELLED ON
THEN-STATE
  X
TO-HAPPEN

IN-STATE
  X
CONDITION
  PA7 OFF?
  PB7 OFF? OR
CAUSES
  YELLED OFF
THEN-STATE
  X
TO-HAPPEN
```

Compare the transitions in the two `ANDGATE`'s to understand the trick in `ANDGATE1`. Notice there is an "action" included in the `ANDGATE1` condition clause. See the **`YELLED OFF`** statement (highlighted in bold) in `ANDGATE1`, not present in `ANDGATE2`? Further notice the same phrase **`YELLED OFF`** appears in the second transition of `ANDGATE2` as the object action of that transition.

TRANSITION COMPARISON

ANDGATE1	**ANDGATE2**	
IN-STATE	IN-STATE	IN-STATE
X	X	X
CONDITION	CONDITION	CONDITION
YELLED OFF		
PA7 ON?	PA7 ON?	PA7 OFF?
PB7 ON? AND	PB7 ON? AND	PB7 OFF? OR
CAUSES	CAUSES	CAUSES
YELLED ON	YELLED ON	**YELLED OFF**
THEN-STATE	THEN-STATE	THEN-STATE
X	X	X
TO-HAPPEN	TO-HAPPEN	TO-HAPPEN

The way this trick worked was by using an action in the condition clause, every time the scheduler ran the chain of machines, it would execute the conditions clauses of all transitions on any active state. Only if the condition was true, did any action of a transition get executed. Consequently, the trick used in `ANDGATE1` caused the action of the second transition to happen when conditionals (only) should be running. This meant it was as if the second transition of `ANDGATE2` happened every time. Then if the condition found the action to be a "wrong" output in the conditional, the action of `ANDGATE1` ran and corrected the situation. The brief time the processor took to correct the wrong output was the "glitch" in `ANDGATE1`'s output.

Now this AND gate, `ANDGATE2`, is just like the hardware AND, except not as fast as most modern versions of AND gates implemented in random logic on silicon. The latency of the outputs of `ANDGATE2` are determined by how many times `ANDGATE2` runs per second. The programmer determines the rate, so has control of the latency, to the limits of the CPU's processing power.

The original `ANDGATE1` serves as an example of what not to do, yet also just how flexible you can be with the language model. Using an action between the `CONDITION` and `CAUSES` phrase is not prohibited, but is considered not appropriate in the paradigm of IsoStructure.

An algorithm flowing to determine a single Boolean value should be the only thing in the condition clause of a transition. Any other action there slows the machine down, being executed every time the machine chain runs.

Most of the time, states wait. A state is meant to take no action, and have no output. They run the condition only to check if it is time to stop the wait, time to take an action in a transition.

The actions we have taken in these simple machines if very short. More complex machines can have very complex actions, which should only be run when it is absolutely necessary. Putting actions in the conditional lengthens the time it takes to operate waiting machines, and steals time from other transitions.

Why was it necessary to have two transitions to do a proper AND gate? To find the answer look at the output of an AND gate. There are two possible mutually exclusive outputs, a "1" or a "0". Once action cannot set an output high or low. One output can set a bit high. It takes a different output to set a bit low. Hence, two separate outputs are required.

ANDOUT

Couldn't we just slip an action into the condition spot and do away with both transitions? Couldn't we just make a "thread" to do the work periodically? Yes, perhaps, but that would break the paradigm. Let's make a non-machine definition. The output of our conditional is in fact a Boolean itself. Why not define:

```
: ANDOUT PA7 ON? PB7 ON? AND IF YELLED ON ELSE YELLED OFF THEN ;
```

Why not forget the entire machine and state mumbo jumbo, and stick `ANDOUT` in the machine chain instead? There are no backwards branches in this code. It has no Program Counter Capture (PCC) Loops. It runs straight through to termination. It would work.

This, however, is another trick you should avoid. Again, why? This code does one of two actions every time the scheduler runs. The actions take longer than the Boolean test and transfer to another thread. The system will run slower, because the same outputs are being generated time after time, whether they have changed or not. While the speed penalty in this example is exceedingly small, it could be considerable for larger state machines with more detailed actions.

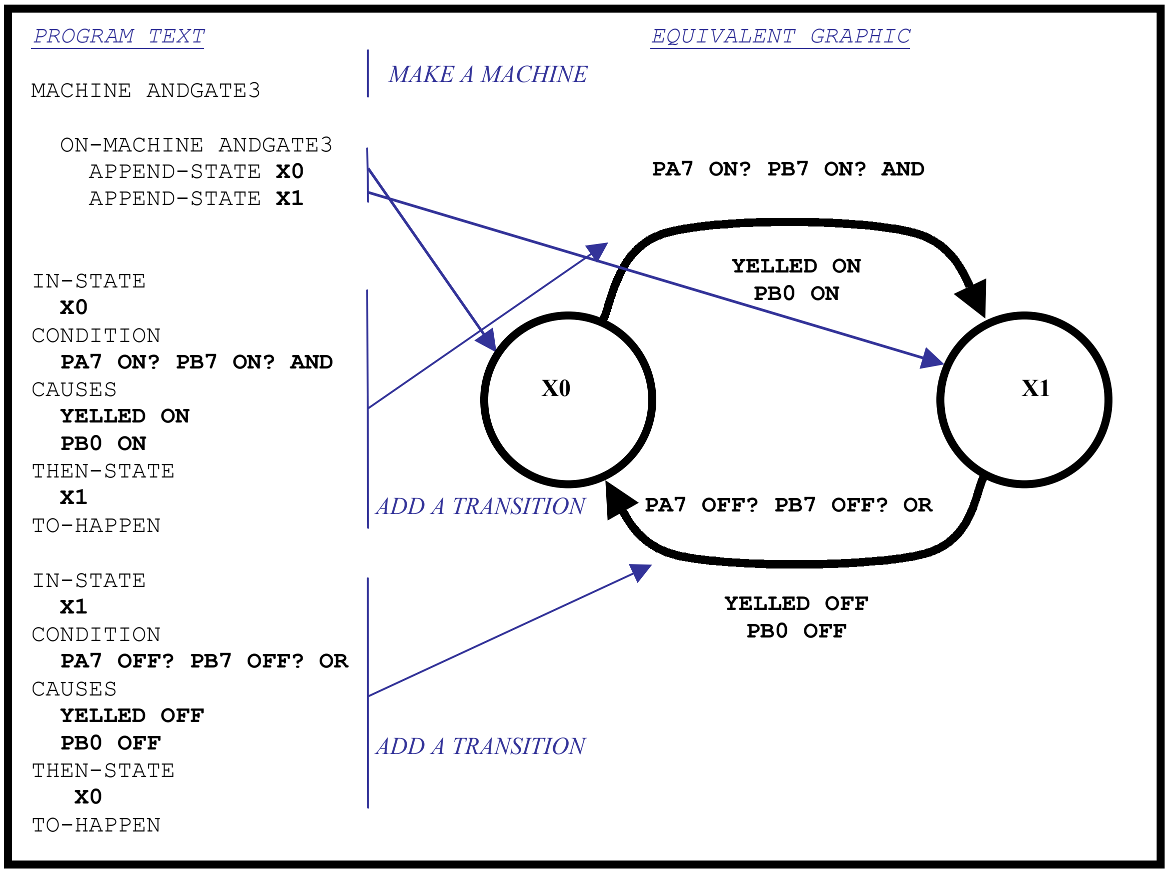
A deeper reason exists that reveals a great truth about state machines. Notice we have used a state machine to simulate a hardware gate. What the AND gate outputs next is completely dependent on what the inputs are next. An AND gate has an output which has no feedback. An AND gate has no memory. State machines can have memory. Their future outputs can depend not only on the inputs present. A state machine's outputs can also depend on the history of previous states. To appreciate this great difference between state machines and simple gates, we must first look a bit further at some examples with multiple states and multiple transitions.

ANDGATE3

We are going to do another AND gate version, `ANDGATE3`, to illustrate this point about state machines having multiple states. This version will have two transitions and two states. Up until now, our machines have had a single state. Machines with a single state in general are not very versatile or interesting. You need to start thinking in terms of machines with many states. This is a gentle introduction starting with a familiar problem. Another change is in effect here. We have previously first written the code so as to make the program small in terms of lines. We used this style to emphasize small program length. From now on, we are going to pretty print it so it reads as easily as possible, instead.

```
( THESE GREY'D TEXT LINES ARE PATCHES FOR V0.2 UPDATE TO V0.3
( ASSUME ON? ALREADY DEFINED

MACHINE ANDGATE3
  ON-MACHINE ANDGATE3
    APPEND-STATE X0
    APPEND-STATE X1

IN-STATE
  X0
```

```
CONDITION
  PA7 ON? PB7 ON? AND
CAUSES
  YELLED ON
  PB0 ON
THEN-STATE
  X1
TO-HAPPEN

IN-STATE
  X1
CONDITION
  PA7 OFF? PB7 OFF? OR
CAUSES
  YELLED OFF
  PB0 OFF
THEN-STATE
  X0
TO-HAPPEN

X0 SET-STATE ( INSTALL ANDGATE3
EVERY 50000 CYCLES SCHEDULE-RUNS ANDGATE3
```



```
MACHINE ANDGATE3

  ON-MACHINE ANDGATE3
    APPEND-STATE X0
    APPEND-STATE X1


IN-STATE
  X0
CONDITION
  PA7 ON? PB7 ON? AND
CAUSES
  YELLED ON
  PB0 ON
THEN-STATE
  X1
TO-HAPPEN

IN-STATE
  X1
CONDITION
  PA7 OFF? PB7 OFF? OR
CAUSES
  YELLED OFF
  PB0 OFF
THEN-STATE
   X0
TO-HAPPEN
```

Notice how similar this version of an AND gate, ANDGATE3, is to the previous version, ANDGATE2. The major difference is that there are two states instead of one. We also added some “spice” to the action clauses, doing another output on PB0, to show how actions can be more complicated.

INTER-MACHINE COMMUNICATIONS

Now imagine ANDGATE3 is not an end unto itself, but just a piece of a larger problem. Now let's say another machine needs to know if both PA7 and PB7 are both high? If we had only one state, it would have to recalculate the AND phrase, or read back what ANDGATE3 had written as outputs. Rereading written outputs is sometimes dangerous, because there are hardware outputs which is cannot be read back. If we use different states for each different output, the state information itself stores which state is active. All an additional machine has to do to discover the status of PA7 and PB7 AND'ed together is check the stored state information of ANDGATE3. To accomplish this, simply query the state this way.

```
X0 IS-STATE?
```

A Boolean value will be returned that is TRUE if either PA7 and PB7 are low. This Boolean can be part of a condition in another state. On the other hand:

```
X1 IS-STATE?
```

will return a TRUE value only if PA7 and PB7 are both high.

STATE MEMORY

So you see, a state machine's current state is as much as an output as the outputs PB0 ON and YELLOW LED ON are, less likely to have read back problems, and faster to check. The current state contains more information than other outputs. It can also contain history. The current state is so versatile, in fact, it can store all the pertinent history necessary to make any decision on past inputs and transitions. This is the deep truth about state machines we sought.

> 9-2 THE FINITE-STATE MODEL -- BASIC DEFINITION
>
> The behavior of a finite-state machine is described as a sequence of events that occur at discrete instants, designated t = 1, 2, 3, etc. Suppose that a machine *M* has been receiving inputs signals and has been responding by producing output signals. If now, at time t, we were to apply an input signal *x(t)* to M, its response *z(t)* would depend on *x(t)*, as well as the past inputs to *M*.
>
> From: SWITCHING AND FINITE AUTOMATA THEORY, KOHAVI

No similar solution is possible with short code threads. While variables can indeed be used in threads, and threads can again reference those variable, using threads and variables leads to deeply nested IF ELSE THEN structures and dreaded spaghetti code which often invades and complicates real time programs.

BOUNCELESS+

To put the application of state history to the test, let's revisit our previous version of the machine BOUNCELESS. Refer back to the code for transitions we used in BOUNCELESS.

```
                STATE Y
IN-STATE          IN-STATE
  Y                 Y
CONDITION         CONDITION
  PA7 OFF?          PB6 OFF?
CAUSES            CAUSES
  GRNLED OFF        GRNLED ON
THEN-STATE        THEN-STATE
  Y                 Y
TO-HAPPEN         TO-HAPPEN
```

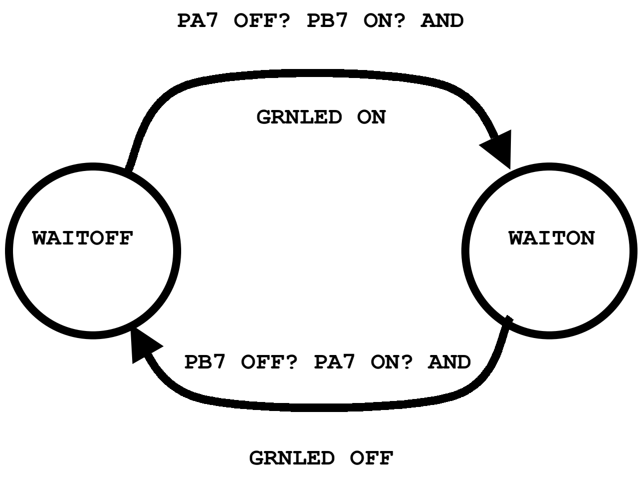
This code worked fine, as long as PA7 and PB6 were pressed one at a time. The green LED would go on and off without noise or bounces between states. Notice however, PA7 and PB6 being low at the same time is not excluded from the code. If both lines go low at the same time, the output of our machine is not well determined. One state output will take precedence over the other, but which it will be cannot be determined from just looking at the program. Whichever transition gets first service will win.

Now consider how BOUNCELESS+ can be improved if the state machines history is integrated into the problem. In order to have state history of any significance, however, we must have multiple states. As we did with our ANDGATE3 let's add one more state. The new states are WAITON and WAITOFF and run our two transitions between the two states.

PROGRAM TEXT

```
MACHINE BOUNCELESS+

  ON-MACHINE BOUNCELESS+
    APPEND-STATE WAITOFF
    APPEND-STATE WAITON


IN-STATE
  WAITOFF
CONDITION
  PA7 OFF? PB7 ON? AND
CAUSES
  GRNLED ON
THEN-STATE
  WAITON
TO-HAPPEN

IN-STATE
  WAITON
CONDITION
  PB7 OFF? PA7 ON? AND
CAUSES
  GRNLED OFF
THEN-STATE
  WAITOFF
TO-HAPPEN
```

EQUIVALENT GRAPHIC



At first blush, the new machine looks more complicated, probably slower, but not significantly different from the previous version. This is not true however. When the scheduler calls a machine, only the active state and its transitions are considered. So in the previous version each time Y was executed, two conditionals on two transitions were tested (assuming no true condition). In this machine, two conditionals on *only* one transition are tested. As a result this machine runs slightly faster.

Further, the new BOUNCELESS+ machine is better behaved. It is truly bounceless, even if both switches are pressed at once. The first input detected down either takes us to its state or inhibits the release of its state. The other input can dance all it wants, as long as the one first down remains down. Only when the original input is released can a new input cause a change of state. In the rare case where both signals occur at once, it is the history, the existing state, which determines the status of the machine.

STATE WAITOFF	**STATE WAITON**
IN-STATE	IN-STATE
WAITOFF	**WAITON**
CONDITION	CONDITION
PA7 OFF? PB7 ON? AND	**PB7 OFF? PA7 ON? AND**
CAUSES	CAUSES
GRNLED ON	**GRNLED OFF**
THEN-STATE	THEN-STATE
WAITON	**WAITOFF**
TO-HAPPEN	TO-HAPPEN

DELAYS

Let’s say we want to make a steady blinker out of the green LED. In a conventional procedural language, like BASIC, C, FORTH, or Java, etc., you’d probably program a loop blinking the LED on then off. Between each loop would be a delay of some kind, perhaps a subroutine you call which also spins in a loop wasting time.

Assembler	**BASIC**	**C JAVA**	**FORTH**
`LOOP1 LDX # 0`	`FOR I=1 TO N`	`While ( 1 )`	`BEGIN`

`LOOP2 DEX`
`      BNE LOOP2`
	`GOSUB DELAY`	`{ delay(x);`	`DELAY`

`LDAA #1`
`STAA PORTA`
`LDX # 0`
	`LET PB=TRUE`	`out(1,portA1);`	`LED-ON`

`LOOP3 DEX`
`      BNE LOOP3`
	`GOSUB DELAY`	`delay(x);`	`DELAY`

`LDAA #N`
`STAA PORTA`
	`Let PB=FALSE`	`out(0,portA1);`	`LED-OFF`

`JMP LOOP1`	`NEXT`	`}`	`AGAIN`

Here’s where IsoMax™ will start to look different from any other language you’re likely to have ever seen before. The idea behind Virtually Parallel Machine Architecture is constructing virtual machines, each a little “state machine” in its own right. But this IsoStructure requires a limitation on the machine, themselves. In IsoMax™, there are no program loops, there are no backwards branches, there are no calls to time wasting delays allowed. Instead we design machines with states. If we want a loop, we can make a state, then write a transition from that state that returns to that state, and accomplish roughly the same thing. Also in IsoMax™, there are no delay loops.

The whole point of having a state is to allow “being in the state” to be “the delay”.

Breaking this restriction will break the functionality of IsoStructure, and the parallel machines will stop running in parallel. If you’ve ever programmed in any other language, your hardest habit to break will be to get away from the idea of looping in your program, and using the states and transitions to do the equivalent of looping for you.

A valid condition to leave a state might be a count down of passes through the state until a 0 count reached. Given the periodicity of the scheduler calling the machine chain, and the initial value in the counter, this would make a delay that didn’t “wait” in the conventional sense of backwards branching.

BLINKGRN

Now for an example of a delay using the count down to zero, we make a machine `BLINKGRN`. Reset your IsoPod™ so it is clean and clear of any programs, and then begin.

```
MACHINE BLINKGRN
  ON-MACHINE BLINKGRN
    APPEND-STATE BG1
    APPEND-STATE BG2
```

The action taken when we leave the state will be to turn the LED off and reinitialize the counter. The other half of the problem in the other state we go to is just the reversed. We delay for a count, then turn the LED back on.

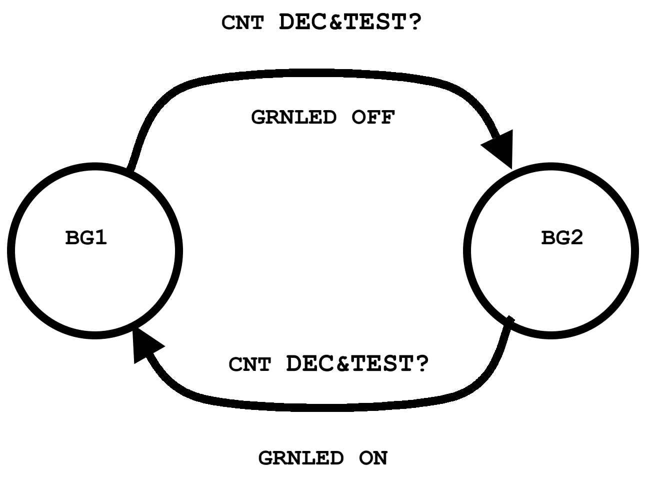
Since we're going to count, we need two variables to work with. One contains the count, the other the initial value we count down from. Let's add a place for those variables now, and initialize them

```
: LOOPVAR <BUILDS HERE P, DUP , , DOES> P@ ;
100 LOOPVAR CNT

: DEC&TEST?
  DUP 1-! DUP @ 0= IF DUP 1 + @ SWAP ! TRUE ELSE DROP FALSE THEN ;

IN-STATE
   BG1
CONDITION
   CNT DEC&TEST?
CAUSES
   GRNLED OFF
THEN-STATE
   BG2
TO-HAPPEN

IN-STATE
   BG2
CONDITION
   CNT DEC&TEST?
CAUSES
   GRNLED ON
THEN-STATE
   BG1
TO-HAPPEN
```

PROGRAM TEXT

```
MACHINE BLINKGRN

  ON-MACHINE BLINKGRN
    APPEND-STATE BG1
    APPEND-STATE BG2

100 0 LOOPVAR CNT

IN-STATE
  BG1
CONDITION
  CNT DEC&TEST?
CAUSES
  GRNLED OFF
THEN-STATE
  BG2
TO-HAPPEN

IN-STATE
  BG2
CONDITION
  CNT DEC&TEST?
CAUSES
  GRNLED ON
THEN-STATE
  BG1
TO-HAPPEN
```

EQUIVALENT GRAPHIC



Above, the two transitions are "pretty printed" to make the four components of a transition stand out. As discussed previously, as long as the structure is in this order it could just as well been run together on a single line (or so) per transition, like this

```
IN-STATE BG1 CONDITION CNT DEC&TEST? CAUSES GRNLED OFF THEN-STATE BG2
TO-HAPPEN

IN-STATE BG2 CONDITION CNT DEC&TEST? CAUSES GRNLED ON THEN-STATE BG1
TO-HAPPEN
```

Finally, the new machine must be installed and tested

```
BG1 SET-STATE ( INSTALL BLINKGRN
EVERY 50000 CYCLES SCHEDULE-RUNS BLINKGRN
```

The result of this program is that the green LED blinks on and off. Every time the scheduler runs the machine chain, control is passed to which ever state `BG1` or `BG2` is active. The `LOOPVAR` created word `CNT` is decremented and tested. When the `CNT` reaches zero, it is reinitialize back to the originally set value, and passes a Boolean on to be tested by the transition. If the Boolean is `TRUE`, the action is initiated. The `GRNLED` is turned `ON` of `OFF` (as programmed in the active state) and the other state is set to happen the next control returns to this machine.

SPEED

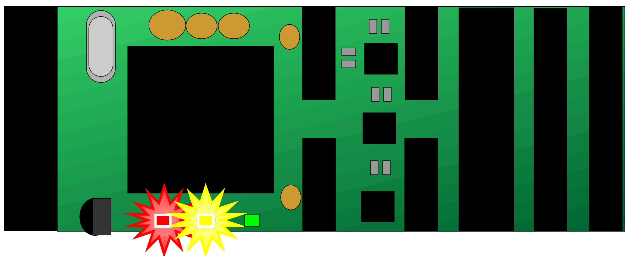
You’ve seen how to write a machine that delays based on a counter. Let’s now try a slightly less useful machine just to illustrate how fast the IsoPod™ can change state. First reset your machine to get rid of the existing machines.

ZIPGRN

```
MACHINE ZIPGRN

ON-MACHINE ZIPGRN
APPEND-STATE ZIPON
APPEND-STATE ZIPOFF

IN-STATE ZIPON CONDITION TRUE CAUSES GRNLED OFF THEN-STATE ZIPOFF
TO-HAPPEN

IN-STATE ZIPOFF CONDITION TRUE CAUSES GRNLED ON THEN-STATE ZIPON
TO-HAPPEN

ZIPON SET-STATE
```

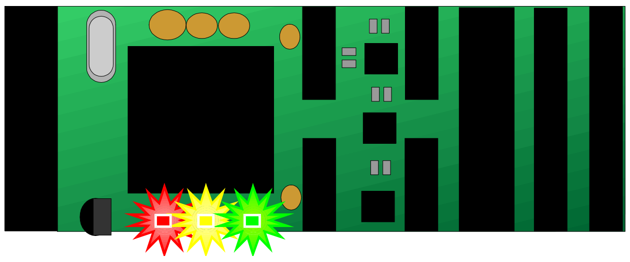
Now rather than install our new machine we’re going to test it by running it “by hand” interactively. Type in:

```
ZPON SET-STATE
ZIPGRN
```



`ZIPGRN` should cause a change in the green LED. The machine runs as quickly as it can to termination, through one state transition, and stops. Run it again. Type:

```
ZIPGRN
```



Once again, the green LED should change. This time the machine starts in the state with the LED off. The always `TRUE` condition makes the transition's action happen and the next state is set to again, back to the original state. As many times as you run it, the machine will change the green LED back and forth.

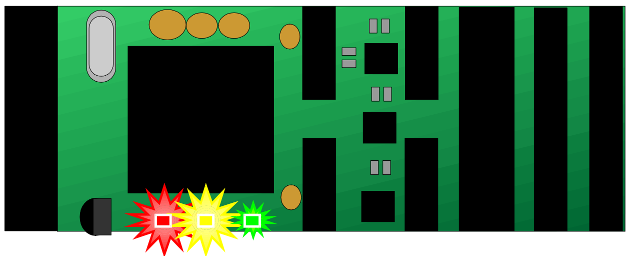
Now with the machine program and tested, we're ready to install the machine into the machine chain. The phrase to install a machine is :

```
EVERY n CYCLES SCHEDULE-RUNS word
```

So for our single machine we'd say:

```
ZIPON SET-STATE
EVERY 5000 CYCLES SCHEDULE-RUNS ZIPGRN
```

Now if you look at your green LED, you'll see it is slightly dimmed.



That's because it is being turned off half the time, and is on half the time. But it is happening so fast you can't even see it.

REDYEL

Let's do another of the same kind. This time lets do the red and yellow LED, and have them toggle, only one on at a time. Here we go:

```
MACHINE REDYEL

ON-MACHINE REDYEL
APPEND-STATE REDON
APPEND-STATE YELON

IN-STATE REDON CONDITION TRUE CAUSES REDLED OFF YELLED ON THEN-STATE
```

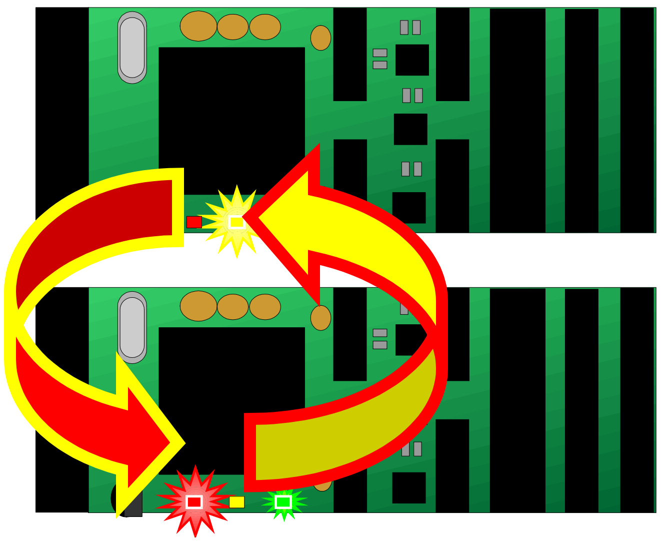
```
YELON TO-HAPPEN

IN-STATE YELON CONDITION TRUE CAUSES REDLED ON YELLED OFF THEN-STATE
REDON TO-HAPPEN
```

Notice we have more things happening in the action this time. One LED is turned on and one off in the action. You can have multiple instructions in an action.

Test it. Type:

```
REDON SET-STATE
REDYEL
REDYEL
REDYEL
REDYEL
```

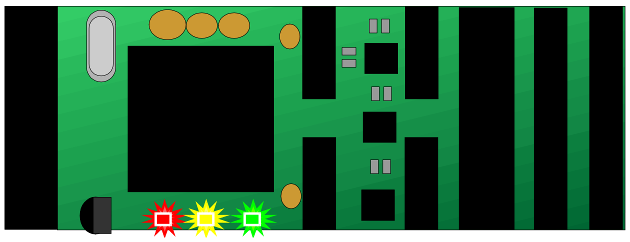
See the red and yellow LED's trade back and forth from on to off and vice versa.



All this time, the `ZIPGRN` machine has been running in the background, because it is in the installed machine chain. Let's replace the installed machine chain with another. So we define a new machine chain with both our virtual machines in it, and install it.

```
MACHINE-CHAIN CHN2
  ZIPGRN
  REDYEL
END-MACHINE-CHAIN

REDON SET-STATE
EVERY 5000 CYCLES SCHEDULE-RUNS CHN2
```

With the new machine chain installed, all three LED's look slightly dimmed.



Again, they are being turned on and off a thousand times a second. But to your eye, you can't see the individual transitions. Both our virtual machines are running in virtual parallel, and we still don't see any slow down in the interactive nature of the IsoPod™.

So what was the point of making these two machines? Well, these two machines are running faster than the previous ones. The previous ones were installed with 50,000 cycles between runs. That gave a scan-loop repetition of 100 times a second. Fine for many mechanical issues, on the edge of being slow for electronic interfaces. These last examples were installed with 5,000 cycles between runs. The scan-loop repetition was 1000 times a second. Fine for many electronic interfaces, that is fast enough. Now let's change the timing value. Redo the installation with the `SCHEDULE-RUNS` command.

The scan-loop repetition is 10,000 times a second.

```
EVERY 500 CYCLES SCHEDULE-RUNS CHN2
```

```
Let's see if we can press our luck.
```

```
EVERY 100 CYCLES SCHEDULE-RUNS CHN2
```

Even running two machines 50,000 times a second in high-level language, there is still time left over to run the foreground routine. This means, two separate tasks are being started and running a series of high-level instructions 50,000 times a second. This shows the IsoPod™ is running more than four hundred thousand high-level instructions per second. The IsoPod™ performance is unparalleled in any small computer available today.

TRINARIES

With the state machine structures already given, and a simple input and output words many useful machines can be built. Almost all binary digital control applications can be written with the trinary operators.

As an example, let's consider a digital thermostat. The thermostat works on a digital input with a temperature sensor that indicates the current temperature is either above or below the current set point. The old style thermostats had a coil made of two dissimilar metals, so as the temperature rose, the outside metal expanded more rapidly than the interior one, causing a mercury capsule to tip over. The mercury moving to one end of the capsule or the other made or broke the circuit. The additional weight of mercury caused a slight feedback widening the set point. Most heater systems are digital in nature as well.

They are either on or off. They have no proportional range of heating settings, only heating and not heating. So in the case of a thermostat, everything necessary can be programmed with the machine format already known, and a digital input for temperature and a digital output for the heater, which can be programmed with trinaries.

Input trinary operators need three parameters to operate. Using the trinary operation mode of testing bits and masking unwanted bits out would be convenient. This mode requires: 1) a mask telling which bits in to be checked for high or low settings, 2) a mask telling which of the 1 possible bits are to be considered, and 3) the address of the I/O port you are using. The keywords which separate the parameters are, in order: 1) `SET-MASK`, 2) `CLR-MASK` and 3) `AT-ADDRESS`. Finally, the keyword `FOR-INPUT` finishes the defining process, identifying the trinary operator in effect.

```
DEFINE <name> TEST-MASK <mask> DATA-MASK <mask> AT-ADDRESS <address> FOR-INPUT
```

Putting the keywords and parameters together produces the following lines of IsoMax™ code. Before entering hexadecimal numbers, the keyword `HEX` invokes the use of the hexadecimal 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 0FB1 FOR-INPUT
DEFINE TOO-HOT?  TEST-MASK 01 DATA-MASK 00 AT-ADDRESS 0FB1 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.

```
DEFINE <name> AND-MASK <mask> XOR-MASK <mask> AT-ADDRESS <address> FOR-OUTPUT
DEFINE <name> CLR-MASK <mask> SET-MASK <mask> AT-ADDRESS <address> FOR-OUTPUT
```

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. PA1 was selected for the heater control signal.

When PA1 is high, or set, the heater is turned on. To make PA1 high, requires PA1 to be set, without changing any other bit of the port. Therefore, a set mask of 02 indicates the next to least significant bit in the port, corresponding to PA1, is to be set. All other bits are to be left alone without being set. A clear mask of 00 indicates no other bits of the port are to be cleared.

When PA1 is low, or clear, the heater is turned off. To make PA1 low, requires PA1 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 02 indicates the next to least significant bit in the port, corresponding to PA1, 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 02 CLR-MASK 00 AT-ADDRESS 0FB0 FOR-OUTPUT
DEFINE HEATER-OFF SET-MASK 00 CLR-MASK 02 AT-ADDRESS 0FB0 FOR-OUTPUT
DECIMAL
```

Only a handful of system words need to be covered to allow programming at a system level, now.